Sviluppare applicazioni web con Django

Marco Beri

SCOPRIRE
il framework pensato per i perfezionisti alle prese con una deadline

SCRIVERE
meno codice possibile, evitando le ridondanze e ottimizzando le prestazioni

"Non ti ripetere mai! Ogni singolo concetto o frammento di dati deve essere presente una – e una sola – volta. La ridondanza è male. La normalizzazione è buona. Per questo il framework deve essere in grado di dedurre dal meno possibile, il massimo possibile."
– Filosofia dei creatori di Django – www.djangoproject.com

APOGEO

La presente copia
è un estratto del volume:

Sviluppare applicazioni web con Django

Autore:
Marco Beri

Socio Unico Giangiacomo Feltrinelli Editore s.r.l.
Via Natale Battaglia 12 – 20127 Milano (Italy)
Telefono: 02289981 – Fax: 0226116334
Email apogeo@apogeonline.com
U.R.L. www.apogeonline.com
ISBN: 978-88-503-2817-8

Scheda libro:
www.apogeonline.com/libri/9788850328178/scheda

Sviluppare applicazioni web con Django

Guida completa

Indice generale

Dedicato al mio caro amico ……
(scrivi il tuo nome)

Introduzione

"E ora… qualcosa di completamente differente!"
Monty Python

Forse non lo sapevate ma questo è un libro di cucina. Se volete una torta cosa fate? Facile: potete andare a comprarla in pasticceria. Oppure potete provare a farvela da soli impastando farina, zucchero, uova e un po' di altre cose. Il secondo approccio potrebbe sembrare più difficile ma se ci pensate bene ha moltissimi vantaggi: potete fare una torta anche a mezzanotte con le pasticcerie chiuse, potete farla come preferite, ma, soprattutto, sapete quello che c'è dentro e come è fatta.
Bene, questo libro vuole insegnarvi a "cucinare" con il framework Django e, se riuscirà nel suo scopo, potrete fare i siti o le applicazioni web che volete, quando volete, come volete. E senza dipendere da nessun pasticciere.

Ah, Django è un framework?

Qualcuno di voi forse si chiederà cos'è un *framework*, termine in voga e spesso usato a sproposito in ambito informatico, proprio perché molto alla moda (in inglese esiste un termine che definisce precisamente questo concetto: *buzzword*, "parola del momento").
L'oggetto di tutti i giorni che più assomiglia a un framework è un tavolo da lavoro con tutti gli attrezzi necessari: non ha uno scopo predefinito, ma fornisce i mezzi per risolvere problemi in un dato ambito. Nel nostro caso questa definizione è assolutamente azzeccata, perché Django è esattamente questo: un ottimo tavolo da lavoro, con tanto di strumenti, per costruire siti dinamici e applicazioni web.

Parti del libro

I libri tecnologici troppo "grossi" spesso deludono il lettore: chi cerca un'introduzione rischia di perdersi, mentre chi cerca un approfondimento si annoia per metà delle pagine.
Per questo motivo, scrivendo questo libro, ho deciso di dividerlo in tre parti: nella prima vedremo le basi di Django costruendo un semplicissimo sito dinamico; nella seconda approfondiremo le sue caratteristiche con maggior dettaglio; infine, nella terza vedremo

una serie di "ricette" per risolvere problemi che spesso si incontrano sviluppando applicazioni web (in fondo non si è mai visto un libro di cucina senza ricette). A seconda della vostra conoscenza di Django potrete passare direttamente alla Parte II o III.

Perché Django?

Adrian Holovaty, uno degli ideatori e autori di Django assieme a Simon Willison e Jacob Kaplan-Moss, è anche un appassionato di musica; se volete, potete vederlo all'opera su YouTube mentre suona la chitarra, all'indirizzo `http://www.youtube.com/adrianholovaty`.
In particolare Adrian è appassionato di gipsy jazz (musica zingara), un genere che è stato reso famoso da un grandissimo chitarrista: Django Reinhardt (Figura I.1). La storia di Reinhardt è avvincente anche per via di un grave incidente che subì a diciotto anni: un incendio distrusse la sua roulotte, rendendo quasi inutilizzabili le ultime due dita della sua mano sinistra. Trovandosi in una situazione che avrebbe interrotto la carriera di qualunque altro chitarrista, Django sviluppò invece una tecnica rivoluzionaria che gli permise comunque di suonare divinamente la sua chitarra.
Adrian ha scelto il nome Django per la sua passione per la musica gipsy. A me però piace pensare che un altro motivo sia anche il fatto che, come questo straordinario musicista riuscì a fare il suo lavoro con il 40% in meno delle dita della mano sinistra, noi possiamo, grazie a Django, fare il nostro lavoro nel 40% in meno del tempo. :-)

Figura I.1 Django Reinhardt.

Cosa è meglio sapere?

Python. È inutile girarci intorno: se volete imparare a cucinare con Django dovete conoscere Python. È l'unico prerequisito veramente indispensabile.
Python è il linguaggio in cui Django è scritto, ma non è solo per questo motivo che dovete conoscerlo. Come vedremo, quasi ogni oggetto in Django è codice Python; lo è perfino il file di configurazione principale. Inoltre a volte può essere interessante, oltre che utile, dare un'occhiata al codice sorgente di Django stesso, che è liberamente scaricabile e disponibile, come vedremo nel Capitolo 1, dedicato all'installazione.

Python

Nell'eventualità che non lo conosciate e che stiate decidendo se vale la pena o meno di impararlo per poter usare Django, provo a stuzzicare il vostro appetito (ricordate: questo è un libro di cucina).
La prima domanda delle FAQ (*Frequently Asked Questions*, le domande più frequenti) sul sito ufficiale di Python `http://www.python.org` è "Cos'è Python?".
Leggiamo assieme la risposta: "Python è un linguaggio di programmazione interpretato, interattivo e orientato agli oggetti. Incorpora al suo interno moduli, eccezioni, tipizzazione dinamica, tipi di dato di altissimo livello e classi. Python combina una eccezionale potenza con una sintassi estremamente chiara. Ha interfacce verso molte chiamate di sistema, oltre che verso diversi ambienti grafici, ed è estendibile in C e in C++. È inoltre usabile come linguaggio di configurazione e di estensione per le applicazioni che richiedono un'interfaccia programmabile. Da ultimo, Python è portabile: può girare su molte varianti di UNIX, sul Mac, sui PC con MS-DOS, Windows, Windows NT e OS/2".
A chi dobbiamo una meraviglia del genere? A un geniale signore olandese che risponde al nome di Guido Van Rossum. Curiosamente Guido non è la traduzione italiana del suo nome, è proprio il suo nome originale in olandese.
Guido, nel lontano Natale del 1989, invece di passare le sue vacanze a decorare l'albero, decise di scrivere un linguaggio che correggesse la maggior parte dei difetti, se non tutti, che secondo lui erano presenti negli altri linguaggi.
Per nostra fortuna Guido Van Rossum era, ed è tuttora, un grande esperto di linguaggi di programmazione e questo ha fatto sì che fin da subito la sua creatura avesse un notevole successo, dapprima tra i colleghi del centro di ricerca dove lavorava in quel periodo e poi, dopo la pubblicazione su USENET nel febbraio del 1991, in tutto il mondo.
Una domanda che spesso sento fare da chi non lo conosce è: "Ma c'è qualcuno che usa Python in ambito professionale?".
La risposta non può che essere "Sì!". IBM, Google, Sun, Hewlett Packard, Industrial Light & Magic (quelli che hanno fatto *Guerre Stellari*), la NASA (quelli che sono andati sulla Luna); perfino in Microsoft usano Python. Non ci credete? Provate a digitare in Google **site:microsoft.com python**.
In questo momento ci sono 9780 risultati e uno di questi si intitola *Open Source at Microsoft* e si occupa di IronPython, la versione di Python per .NET.
Forse sarebbe più corretto domandarsi chi ancora non lo usa...

Cosa ci aspetta per pranzo?

Prima di cominciare a occuparci seriamente di Django (finalmente, direte voi) voglio tranquillizzare quelli che, arrivati sino a qua, possono temere che questo framework sia fatto per chi vuole sì fare tutto da solo, ma reinventando la ruota ogni volta.
Niente di più sbagliato! Django è stato creato da programmatori Python che, come tutti i veri programmatori, non possono che essere fondamentalmente pigri e quindi non amano rifare lo stesso lavoro due volte.
Date uno sguardo alla Figura I.2, dove è mostrata una gradevolissima interfaccia web per l'inserimento dell'anagrafica di un autore da usare in un articolo, a sua volta appena inserito.

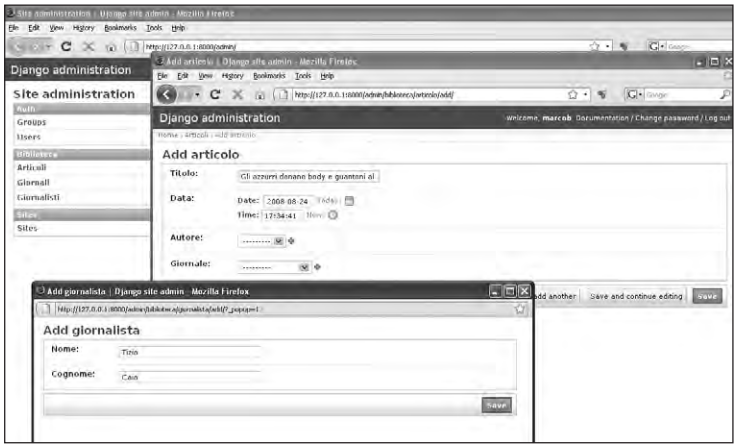

Figura I.2 Un esempio dell'interfaccia di amministrazione di Django.

Partendo da zero, per arrivare sino a qui e con solo Python e Django installati sul mio computer, ho modificato 5 righe in due file di configurazione, ho creato un file per definire il mio database (15 righe) e ho eseguito uno script di Django per creare il progetto e il database (SQLite – `http://www.sqlite.org` – già compreso in Python).
Quindi ho avviato il server integrato in Django per effettuare i test sul progetto. Tempo totale: circa 10 minuti. Tutto qui.
Già mi pare di sentire i più esigenti e attenti obiettare "ma è in inglese!". Benissimo: modifichiamo un'altra riga del file di configurazione ed ecco nella Figura I.3 il nuovo pannello di amministrazione completamente in italiano.

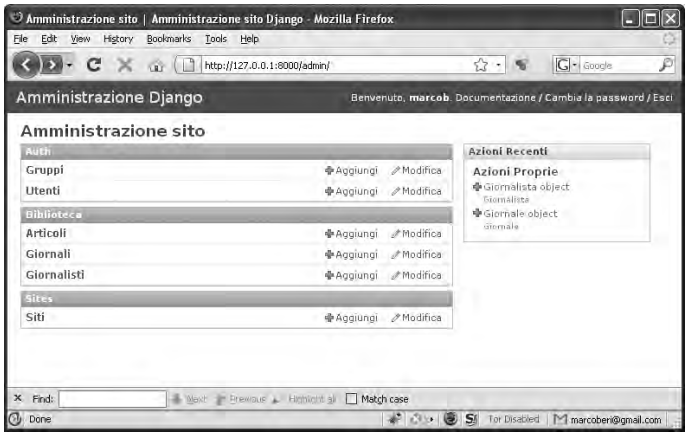

Figura I.3 L'interfaccia in italiano.

Giusto per la cronaca, Django ha a già disposizione, per le stringhe visibili all'utente, traduzioni in 49 lingue; visitate l'indirizzo `http://code.djangoproject.com/browser/django/trunk/django/conf/locale`.
Bene! E adesso cominciamo a cucinare...

Parte I

Gli antipasti

"Va bene! Ma a parte le fognature, vino, medicina, istruzione,asini pubblici in orario, ordine pubblico, irrigazione, strade, spiagge libere non inquinate, bilancio dei pagamenti in attivo, che cosa hanno fatto i Romani per noi?"
Brian di Nazareth – Monty Python

Django è composto da tanti elementi, ben separati l'uno dall'altro, ognuno preposto a una funzione ben precisa. In questa prima parte del libro conosceremo i "mattoncini" più importanti, che ci permetteranno di costruire un semplice sito dinamico.
Affronteremo via via diversi aspetti fondamentali come l'installazione del framework, il database, il pannello di amministrazione, la gestione degli URL, i template e i form.
Anche se vedremo solo un piccolo insieme di funzionalità, rispetto a quello che ci aspetta nella seconda e terza parte di questo libro, sono sicuro che, come antipasto, sarà più che sufficiente per immaginare il resto del banchetto.

In questa parte

Capitolo 1

L'installazione, a.k.a. "Hello world"

In questo capitolo

- **Installazione di Python**
- **Installazione di Django**
- **"Hello world"**
- **Creazione del progetto**
- **Creazione di un URL**
- **Creazione della vista**

"Ecco la macchina che fa PING!"
Il senso della vita – Monty Python

Spesso il primo capitolo di un libro informatico (va bene, lo confesso: in questo libro si parlerà, purtroppo, più di informatica che di cucina) spiega l'installazione dell'oggetto trattato nei capitoli seguenti.
Questo libro non farà eccezione e partirà proprio dall'installazione di Django (dopo un brevissimo excursus su Python).
Per ottenere il nostro scopo, creeremo anche un piccolo progetto che ci servirà nei capitoli seguenti per scoprire le funzionalità di Django (quindi, anche se avete già installato Django, mi spiace, ma non potete saltare al prossimo capitolo :-)).
Un paio di spiegazioni veloci sul titolo in cima a questa pagina.

- "Hello world" è la frase che viene quasi sempre usata come primo esempio in qualunque oggetto informatico che produca un output (un linguaggio di programmazione, un framework e così via). È curioso scoprire che è stata usata per la prima volta nel famosissimo libro *Il linguaggio C*, scritto nel 1974 da Brian Kernighan e Dennis Ritchie (praticamente una bibbia per i programmatori C).
- *a.k.a.* è l'acronimo dell'espressione inglese *also known as* ("anche conosciuto come").

Installazione di Python

Come abbiamo detto nel capitolo precedente, conoscere Python costituisce un prerequisito per l'uso di Django. Non possiamo permetterci quindi di spendere molto tempo per spiegare la sua installazione che, molto probabilmente, abbiamo già eseguito.
Ad ogni modo, se vogliamo imbarcarci nell'impresa (stimolante, difficile, ma non certo titanica) di imparare Python insieme a Django, la prima cosa che dobbiamo fare è scaricarlo da Internet e installarlo sul nostro PC.
Innanzitutto dobbiamo verificare che Python non sia già installato sul nostro *ferro* (non tutti sanno che la traduzione letterale di *hardware* è "ferramenta"). Eseguiamo da linea di comando il seguente... comando:

```
python -version
```

Se il risultato è una versione superiore o uguale a 2.5 (e inferiore a 3), non dovete fare altro e potete saltare al prossimo paragrafo. Se il risultato è superiore o uguale 2.3, vi consiglio di installare comunque una versione aggiornata (in questo momento è la 2.6) perché così avrete anche, incluso nel prezzo, il database SQLite che useremo nei prossimi capitoli.
Se l'output del comando è `command not found` o `comando sconosciuto` o comunque qualcosa del genere, dovrete proseguire nella lettura di questo paragrafo.

> **MA PYTHON È PREINSTALLATO SOLO SU GNU/LINUX O MAC OSX**
>
> Questa affermazione poteva essere vera sino a qualche tempo fa. Oggigiorno esistono diverse marche diffuse che non posso nominare (una di tre lettere maiuscole e una di due) che lo preinstallano sui loro personal computer, venduti con il solito sistema operativo.

In Internet troviamo la "casa" di Python all'indirizzo `http://python.org`. Scarichiamo la versione più adatta al nostro sistema operativo, cercando nella pagina `http://python.org/download`.
Se abbiamo fatto tutto nella maniera corretta, dobbiamo essere in grado di eseguire il comando `python` per arrivare al suo fatidico prompt `>>>` (i tre caratteri `>`).
Possiamo vedere il prompt di Python nella Figura 1.1 su GNU/Linux (distribuzione Kubuntu) e nella Figura 1.2 su Windows.
Se vi state domandando perché sembra più carina la Figura 1.1, non preoccupatevi: significa che avete buon gusto.

Installazione di Django

Se `http://python.org` è l'indirizzo della casa di Python, in via `http://www.djangoproject.com` abita Django.
Per installare il nostro framework preferito (se non la ripetessi così spesso che *buzzword* sarebbe?), dobbiamo scaricare il pacchetto con i sorgenti dell'ultima versione ufficiale che troviamo all'indirizzo `http://www.djangoproject.com/download/`.
Attualmente la versione ufficiale è la 1.0 e il corrispondente file da scaricare è quindi `Django-1.0.tar.gz`.

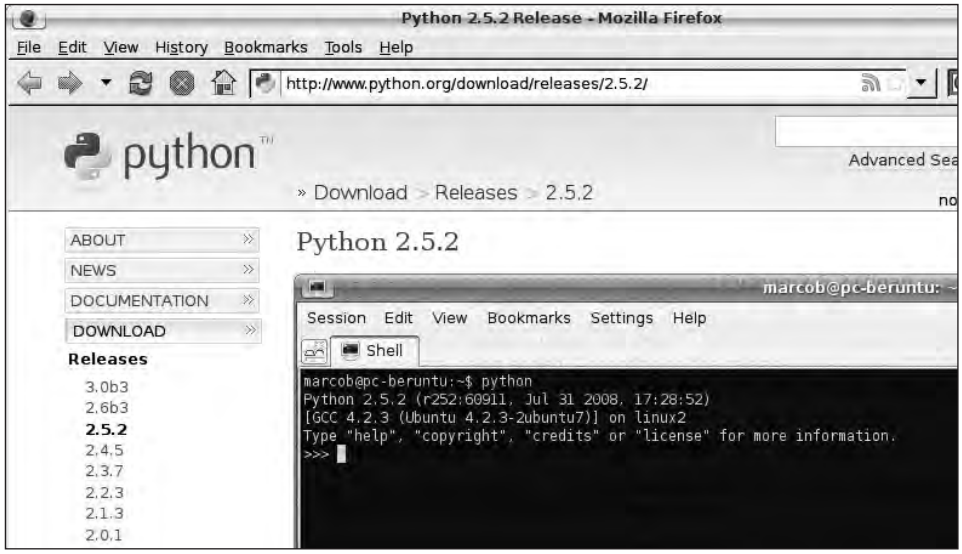

Figura 1.1 Python su GNU/Linux.

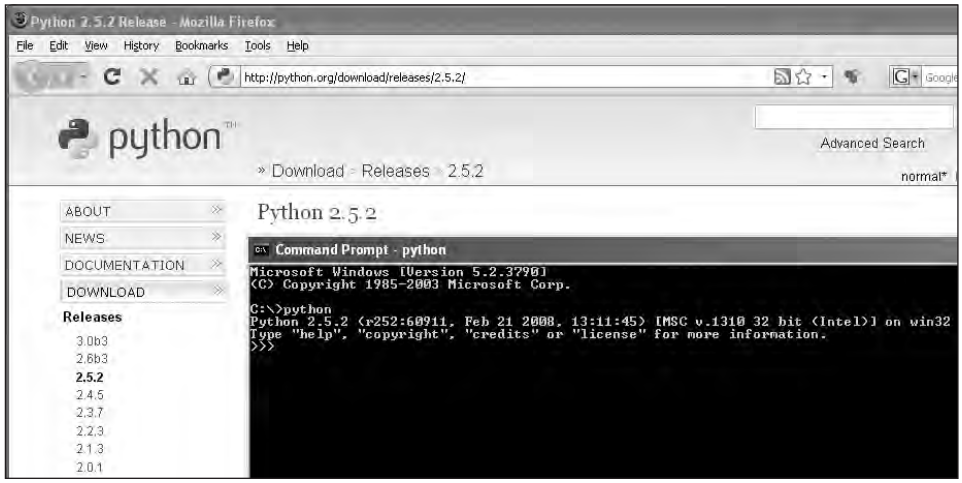

Figura 1.2 Python su Windows.

COS'È UN FILE TAR.GZ?

Per chi usa GNU/Linux un file con estensione `tar.gz` non è niente di strano, anzi, è la normalità. Leggermente diversa è la situazione per chi usa Windows. Esistono programmi gratuiti in grado di decomprimere questi file, come per esempio Tugzip (lo troviamo su `http://www.tugzip.com`). I più *puri* possono anche usare direttamente le versioni per Windows dei programmi gzip e tar (Google ci può aiutare a trovarli) digitando questo esoterico comando:

```
gzip -d < Django-1.0.tar.gz | tar xvf -
```

L'operazione di decompressione del file scaricato creerà una directory chiamata `Django-1.0` (ovviamente al posto di `1.0` potremmo, in futuro, avere un diverso numero di versione).
Entriamo in questa directory e digitiamo il comando `python setup.py install`, usato per l'installazione della maggior parte delle librerie Python. In GNU/Linux (o Mac OSX o qualunque altro sistema UNIX) il comando completo è:

```
sudo python setup.py install
```

SUDO?

Per quelli tra voi che non conoscono Unix: il comando `sudo` indica un'azione da compiere come superuser. Infatti `sudo` non indica che la paura per l'istruzione che stiamo per eseguire ci facendo sudare, ma sta per *Super User DO* ("Fai come superuser").

Al termine della lunga serie di messaggi, che ci informano dello stato di avanzamento dell'installazione, possiamo testare la corretta installazione di Django direttamente dal prompt di Python (Figura 1.3):

```
>>> import django
>>> django.VERSION
(1, 0, 'final')
>>>
```

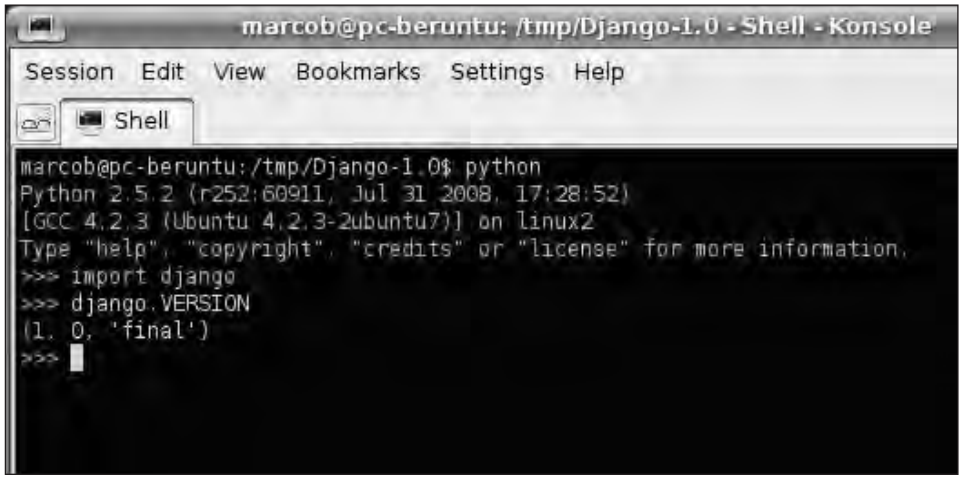

Figura 1.3 Django correttamente installato.

IMPORT DJANGO

Questo rapido test ci da la conferma di un'importante informazione: Django è a tutti gli effetti un modulo Python e, come tutti i moduli Python, può essere usato con il comando `import` direttamente dal prompt.

Se qualcosa è andato storto e non riusciamo a eseguire il test precedente in maniera corretta, non preoccupiamoci più di tanto. Per fortuna esiste in Internet una fonte di aiuto

inesauribile come la comunità italiana degli utilizzatori di Django: visitate l'indirizzo `http://groups.google.com/group/django-it`.
Sarà sufficiente porre le domande con cortesia e chiarezza per essere sorpresi dal numero di persone competenti che saranno disposte ad aiutarci.

> **INSTALLAZIONE DELLA VERSIONE IN SVILUPPO**
>
> Django è stato reso disponibile per la prima volta al pubblico nell'estate del 2005 e quasi da subito è stato usato in ambienti di produzione. Nonostante questo, la versione 1.0 ufficiale ha visto la luce solo nel settembre 2008. Questo ha fatto sì che spesso, negli anni passati, gli utilizzatori desiderassero scaricare e testare i sorgenti aggiornati con gli ultimi sviluppi e correzioni. Il sorgente di sviluppo è infatti liberamente disponibile e può essere scaricato con il programma Subversion. All'indirizzo `http://code.djangoproject.com/` troviamo sia il sorgente sia tutte le istruzioni del caso, se vogliamo fare un giro con l'ultimissima versione.

"Hello world"

Se il test del paragrafo precedente ha avuto successo, abbiamo installato correttamente Django e potremo quindi passare al prossimo capitolo.
Prima, però, dobbiamo ottenere il già promesso "Hello world". Per farlo dobbiamo fare alcune operazioni che probabilmente ci sembreranno poco comprensibili. Non preoccupiamoci troppo: nei prossimi capitoli tutto sarà più chiaro (a patto, ovviamente, che io sia riuscito nel mio intento).

Creazione del progetto

La prima operazione da effettuare è la creazione di un progetto. L'installazione di Django ha creato nel nostro sistema uno script Python dal nome `django-admin.py`.
Se da linea di comando lo digitiamo seguito dall'opzione `--version` dovremmo ottenere questo risultato:

```
$ django-admin.py --version
1.0-final-SVN-unknown
```

Naturalmente al posto del `$` potrà esserci un prompt diverso (per esempio `C:>`).

> **COMMAND NOT FOUND**
>
> Se riceviamo un messaggio di errore di comando non trovato, proviamo a eseguire direttamente il comando che dovrebbe trovarsi nella directory `/usr/bin` (GNU/Linux, Mac OS X) o nella directory `C:\python25\scripts` (Windows). Per esempio:
>
> ```
> C:> python C:\python25\scripts\django-admin.py –version
> 1.0-final-SVN-unknown
> ```

L'argomento per creare un progetto è `startproject` seguito dal nome del progetto, quindi il comando che ci serve è:

```
$ django-admin.py startproject hello
```

L'esecuzione creerà nella directory corrente una sottodirectory avente lo stesso nome che abbiamo dato al progetto, con all'interno i seguenti file:

```
__init__.py
manage.py
settings.py
urls.py
```

Per essere sicuri che tutto abbia funzionato, spostiamoci nella directory `hello` e verifichiamo la loro presenza con il comando `ls` o con il comando `dir` (dovremmo sapere quale di questi due è quello corretto per il nostro sistema, giusto?).

```
$ cd hello
$ ls
__init__.py manage.py settings.py urls.py
```

Django fornisce già un server adatto per i test del nostro sito. Per eseguirlo dobbiamo usare il comando `manage.py` con l'argomento `runserver`:

```
$ python manage.py runserver
```

Se abbiamo fatto tutto bene, apparirà il seguente testo:

```
Validating models...
0 errors found

Django version 1.0-final-SVN-unknown, using settings
                                          'hello.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

L'ultima riga ci indica come interrompere il server di test: premendo contemporaneamente i tasti Ctrl e Break ne termineremo l'esecuzione.
Se così è stato, possiamo aprire con un browser l'URL `http://127.0.0.1:8000/`; nella Figura 1.4 vediamo la schermata di default del server di test di Django.

> **L'URL O LA URL?**
>
> Nel caso ve lo stiate domandando, URL è l'acronimo di *Uniform Resource Locator* (localizzatore uniforme di risorse) e quindi vuole l'articolo maschile (come il genere di localizzatore). Così almeno sostiene l'Accademia della Crusca (`http://www.accademiadellacrusca.it/faq/faq_risp.php?id=5469&ctg_id=93`).

Creazione di un URL

Nel nostro progetto abbiamo bisogno di stabilire un URL per la stampa di "Hello World". Con un notevole sforzo di immaginazione scegliamo `http://127.0.0.1:8000/hello/`. In particolare, la parte di URL più significativa che ci interessa è quella che segue il nome del nostre server e quindi esclusivamente `hello/`.
Una delle caratteristiche più notevoli di Django è la sua capacità di disaccoppiare un URL dalla corrispondente risorsa. In altre parole possiamo scegliere liberamente l'indirizzo che desideriamo, senza preoccuparci di `.php`, `.aspx`, `.jsp` e altre amenità del genere.
Il file preposto è `urls.py`; apriamolo in modifica con il nostro editor preferito (Vi, Vim, Emacs, Xemacs, TextMate, Blocco note, IDLE, CAT, Copy Con e così via) e dovremmo vedere il testo della Figura 1.5.

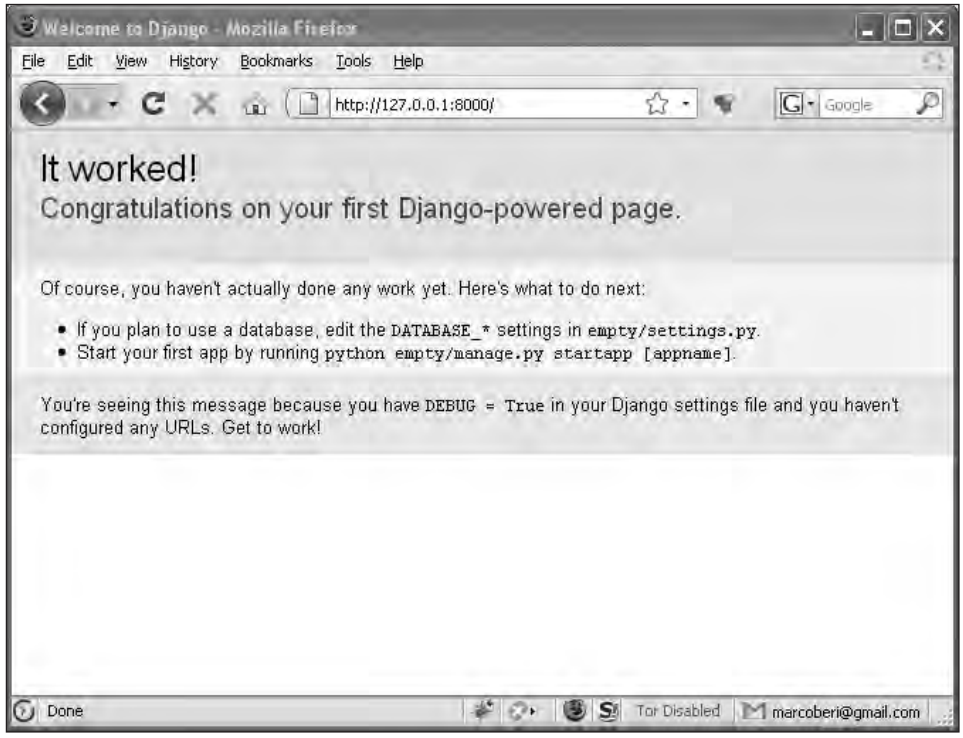

Figura 1.4 La schermata di default del server di test di Django.

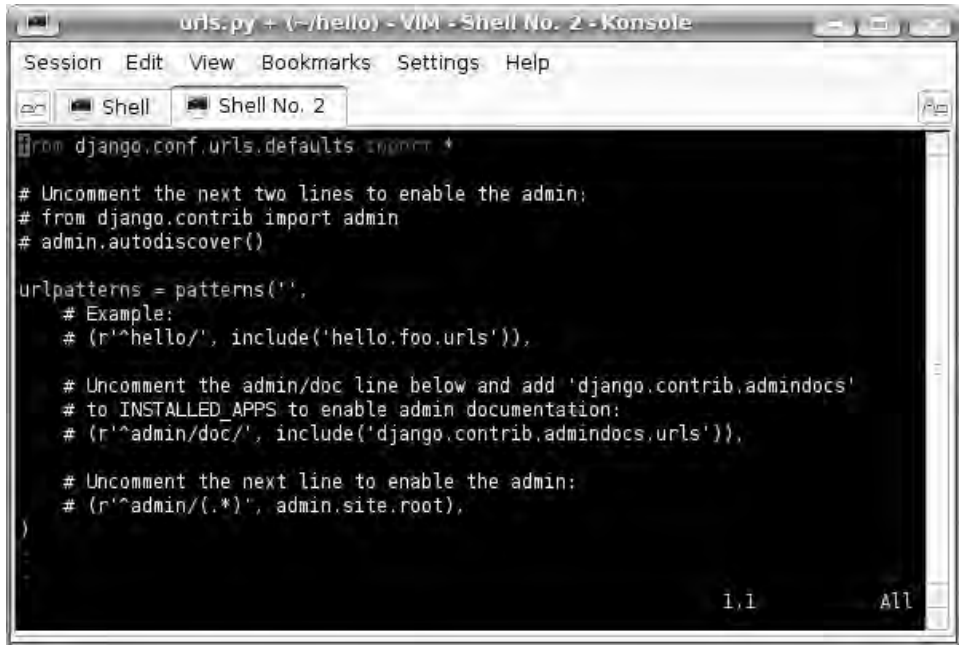

Figura 1.5 L'originale urls.py.

BLOCCO NOTE O IDLE

Blocco note in realtà non è proprio una buona scelta, visto che non ci permette di personalizzare il comportamento del tasto di tabulazione (il Tab per intenderci). Questo potrebbe crearci qualche problema nella modifica di sorgenti Python già esistenti. Per dire tutta la verità, personalmente ho qualche dubbio anche sul sistema operativo che ci sta intorno, ma questa è tutta un'altra storia. Invece IDLE è l'editor che spesso viene installato assieme a Python. IDLE è l'acronimo di *Integrated DeveLopment Environment* (ambiente di sviluppo integrato) oltre che richiamare il nome di Eric Idle, uno dei componenti dei Monty Python.

Una raccomandazione importante a proposito della modifica dei sorgenti in Python: ricordiamoci che con questo eccezionale linguaggio *whitespace matters!* (lo spazio conta!). Non per niente l'indentazione del codice è una regola sintattica in Python.

WHITESPACE MATTERS

Una bellissima t-shirt che spesso si vede alla conferenze di Python recita "Python: programming the way Guido indented it" ("Python: programmare alla maniera indentata da Guido"), battuta che fa leva sul gioco di parole tra *indent* (indentare) e *invent* (inventare) o anche *intend* (intendere).

Inseriamo come ultimo parametro passato alla funzione `patterns` il seguente valore:

```
(r'^hello/$', "views.hello"),
```

Il primo elemento della tupla, `r'^hello/$'` è un'espressione regolare che individua l'URL `hello/` (nelle espressioni regolari il carattere `^` indica l'inizio della riga, mentre il `$` la sua fine). Il secondo elemento, invece, indica a Django la funzione da richiamare quando l'URL viene richiesto dall'utente.

COSA SONO LE ESPRESSIONI REGOLARI?

Sulla copertina di un famosissimo (magari!) libro della collana Pocket di questo stesso editore, che tratta proprio le espressioni regolari, è scritto "Espressioni regolari: inventate nel 1950 ma ancora oggi insostituibili". Il primo capitolo inizia così: "Le espressioni regolari sono oggetti che ci aiutano a cercare qualcosa nelle stringhe che usiamo nei nostri programmi ed eventualmente sostituire parti di esse". In Django questi oggetti così antichi, ma allo stesso tempo così utili e potenti, sono usati per cercare, individuare e suddividere URL completi o loro parti. Curiosamente, nella stessa collana Pocket c'è anche un altro famosissimo (magari!) libro che tratta di Python che, con le espressioni regolari, è appunto l'altra tecnologia alla base di Django.

A questo punto il testo dovrebbe essere quello della Figura 1.6. Salviamo il file e torniamo alla linea di comando.

Creazione della vista

Una *vista* non è altro che una semplice funzione Python, che in Django produce i dati che vengono poi visualizzati dall'utente, perlopiù con un browser.

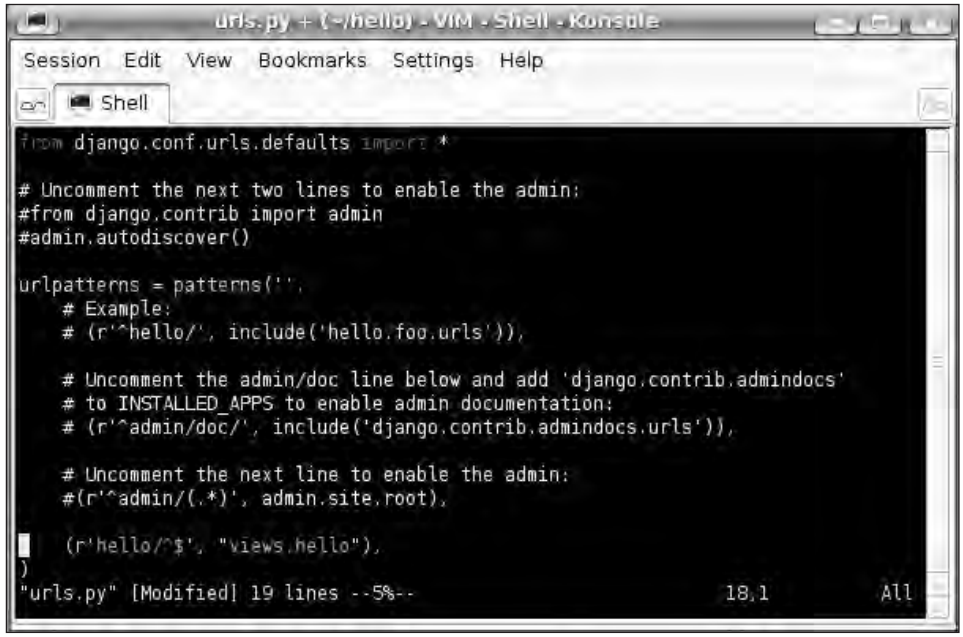

Figura 1.6 urls.py modificato.

Nel passo precedente, inserendo nel file `urls.py` la stringa `"views.hello"`, abbiamo individuato come vista la funzione `hello` del modulo `views.py`.
Creiamo quindi, nella stessa directory dove ci sono gli altri file del progetto, il file `views.py` e inseriamo il seguente testo:

```
from django.http import HttpResponse
def hello(request):
return HttpResponse("Hello world")
```

La vista `hello`, come possiamo vedere nel codice qui sopra, non fa altro che restituire un oggetto `HttpResponse`.
Salviamo il file e torniamo alla linea di comando, perché abbiamo terminato i file da modificare.
Come abbiamo già visto in precedenza, Django fornisce già un server di test, quindi proviamo ad avviarlo nuovamente:

```
$ python manage.py runserver
```

Bene, ora possiamo aprire con un browser l'URL `http://127.0.0.1:8000/hello/` e, *lo and behold!*, ecco il nostro agognato `Hello world`.

> **LO AND BEHOLD!**
>
> Il significato di questa frase, che è la contrazione della versione arcaica estesa *Look! Behold!* ("Guarda! Vedi!"), segnala qualcosa di importante, degno di attenzione. Scrivendolo spero di risparmiarvi, nel caso vi capitasse di trovarlo su qualche sito, il tempo che ho perso io per capirne il significato quando l'ho incontrato per la prima volta in una pagina di documentazione in inglese.

Nella Figura 1.7 vediamo in primo piano il browser e sullo sfondo il server che segnala la richiesta appena servita con il seguente messaggio:

```
[03/Oct/2008 18:57:20] "GET /hello/ HTTP/1.1" 200 11
```

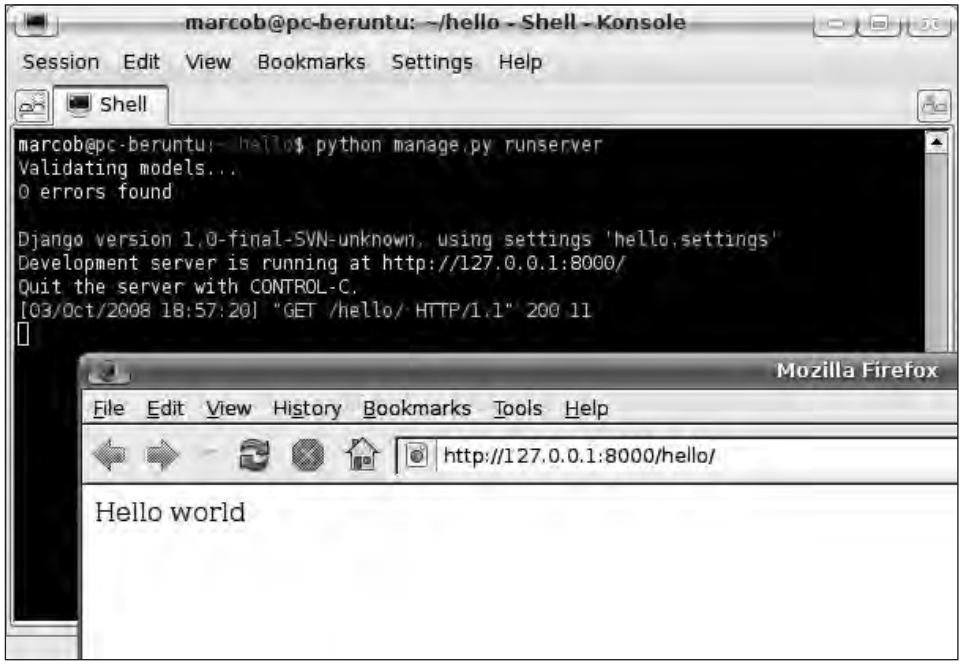

Figura 1.7 Hello world!

Riepilogo

In questo capitolo abbiamo visto:

- come installare Django;
- come creare un progetto (con il comando `django-admin.py startproject` <*nomeprogetto*>);
- come avviare il server di sviluppo (`python manage.py runserver`);
- come abbinare un URL a una funzione Python (nel file `urls.py`);
- infine, come scrivere questa funzione (nel file `views.py`).

Capitolo 2

Il database

In questo capitolo

- **Cosa significa ORM?**
- **Cosa significa MTV?**
- **Creiamo un'applicazione**
- **Creiamo un modello**
- **Inseriamo i dati**
- **Interroghiamo il database**
- **I test**

"E tante grazie per l'oro e per l'incenso, ma per la mirra non dovete disturbarvi la prossima volta, d'accordo?"
Brian di Nazareth – Monty Python

Tranne poche eccezioni, quando scriviamo un programma o realizziamo un sito, dobbiamo occuparci di come e dove salvare dei dati. La risposta è semplice: usiamo un database.
Ma subito dopo altre domande sorgono spontanee: "Che tipo di database? Come ci scrivo i dati? E come faccio a scriverli se uso degli oggetti organizzati in classi?".
Django cerca di risolvere questi (e altri) problemi con il suo ORM, che viene usato pesantemente nel paradigma di programmazione MTV.
Prima che pensiate che sto dando i numeri, o meglio, le lettere, vediamo nei prossimi due paragrafi il significato di queste due sigle.
Solo un pizzico di teoria prima della tanto agognata pratica.

Cosa significa ORM?

ORM è l'acronimo di *Object-Relational Mapping* (mappatura tra oggetti e database relazionali).
In parole povere un ORM cerca di costruire un ponte tra due mondi in apparenza inconciliabili: i database relazionali e la programmazione a oggetti. Attraverso questo ponte gli oggetti creati con un linguaggio di alto livello (per esempio Python) vengono salvati in un database relazionale (per esempio PostgreSQL, MySQL o Oracle).

Percorrendolo nel senso inverso, invece, i dati presenti nelle tabelle di un database relazionale diventano oggetti di un linguaggio di alto livello.
L'uso di un ORM, oltre a permettere questa sorta di *trasformazione* dei dati, porta con sé altri vantaggi non trascurabili, come l'astrazione dal database scelto o la possibilità di interrogare i dati usando la sintassi di un linguaggio di alto livello, piuttosto che quella, certamente meno evoluta, di SQL (acronimo universalmente noto per *Structured Query Language*).
Per capire meglio cos'è un ORM, vediamo un piccolo esempio di quello che affronteremo in dettaglio nel seguito di questo capitolo.
Immaginiamo di avere una tabella di un database contenente dei libri e di voler estrarre tutti quelli che hanno nel loro titolo la parola "aquiloni".
Questa è la richiesta effettuata con l'ORM di Django:

```
Libro.objects.filter(titolo__contains='aquiloni')
```

Il risultato di questa richiesta è una lista di oggetti `Libro`. Non dobbiamo preoccuparci di alcuna conversione: abbiamo già a disposizione il tipo di dati di alto livello che ci interessa quando programmiamo in Python.
La stessa richiesta in SQL nativo sarebbe invece:

```
SELECT * FROM LIBRO WHERE titolo LIKE '%aquiloni%';
```

E come passiamo questa richiesta SQL al nostro database? Cosa otteniamo? Come possiamo convertire quello che otteniamo in oggetti da usare in Python?
Comunque non sono solo questi i problemi che l'ORM di Django risolve. Con il filtro che abbiamo usato nell'esempio precedente, otteniamo solo i libri che contengono al loro interno la stringa "aquiloni" in minuscolo. Se volessimo estrarre anche i titoli con la stringa "Aquiloni" o "AQUILONI" o "aQuILonI" (non si sa mai), potremmo sostituire il filtro nella prima richiesta, cambiando semplicemente `contains` in `icontains`:

```
Libro.objects.filter(titolo__icontains='aquiloni')
```

La "i" di `icontains` sta per *ignore case* (ignora maiuscole e minuscole). Tradurre la query SQL, invece, non è altrettanto semplice. Alcuni database hanno l'operatore `ILIKE` ma altri no, per cui dovremmo forse usare una funzione `UPPER`, `UPPERCASE` o `UCASE`. Per usare una query SQL nativa dobbiamo infatti essere sicuri di scegliere il *dialetto* corretto per il tipo di database che abbiamo scelto. Con l'ORM di Django, invece, non dobbiamo preoccuparci: uno dei suoi compiti, infatti, è anche quello di tradurre le nostre richieste in SQL corretto per il database che abbiamo scelto, permettendoci di cambiarlo senza dover fare lunghi e noiosi adattamenti.

Cosa significa MTV?

Sì, è vero: ancora un altro acronimo... però portiamo pazienza: per un po' non ne vedremo altri e, inoltre, questo è forse il più importante di tutti quelli che riguardano Django.
Lo schema di programmazione che Django abbraccia, o meglio definisce, prende il nome MTV dalle iniziali delle sue tre componenti principali: *Model*, *Template*, *View* . Queste componenti mirano a tenere separati su tre livelli di astrazione diversi i dati (*Model*), le funzioni che agiscono sui dati (*View,* "vista" da qui in avanti nel testo) e come questi vengono presentati all'utente (*Template*).

Un framework che prende il nome da un grandissimo chitarrista non poteva che inventare un pattern di programmazione chiamato come una famosa emittente televisiva musicale.
In questo capitolo ci occuperemo in particolare del livello dei dati, o *Model*.

> **MVC**
>
> Qualcuno avrà già sentito parlare del pattern di programmazione MVC (*Model View Controller*). Manterrò la promessa di non spiegare in dettaglio un altro acronimo, ma ne approfitto per manifestare il mio pieno appoggio a quanto dicono gli autori di Django a proposito degli acronimi: "Alla fine della giornata, naturalmente, tutto si riduce a fare in modo che le cose funzionino. E, indipendentemente da quali nomi usiamo, Django fa in modo che tutto funzioni, nel modo che ci appare più logico" (`http://docs.djangoproject.com/en/dev/faq/general/#mtv`).

Creiamo un'applicazione

Nel primo capitolo abbiamo scritto un progetto minimale, che non faceva altro che visualizzare `Hello world`. Possiamo ora creare al suo interno un'applicazione che contenga il nostro primo modello.

> **COS'È UN'APPLICAZIONE?**
>
> Un'applicazione in Django, in poche e difficili parole, è un insieme minimo di funzionalità coerenti e autocontenute. In parole più facili, un'applicazione è un insieme di funzionalità utili a uno scopo ben preciso e che non è collegata a (e non ha bisogno di) altre applicazioni per essere usata. Molto spesso in Django un progetto può essere composto da un numero molto elevato di piccole applicazioni, probabilmente molte delle quali scelte tra quelle incluse nell'installazione del framework.

Posizioniamoci nella directory del progetto `hello` ed eseguiamo il seguente comando:

```
python manage.py startapp libreria
```

Se non ci sono stati errori, nella nostra directory principale `hello` abbiamo ora una sottodirectory `libreria`, corrispondente all'applicazione che abbiamo creato, e che contiene i seguenti file:

```
__init__.py
models.py
views.py
```

Adesso dobbiamo indicare a Django la presenza dell'applicazione `libreria` e che la vogliamo usare nel nostro progetto. Inoltre dobbiamo scegliere il tipo di database dove scrivere i nostri dati.
Molto più facile a farsi che a dirsi: basta infatti modificare in tutto tre sole righe del file `settings.py`.
Apriamolo con il nostro editor preferito e cerchiamo queste righe:

```
DATABASE_ENGINE = ''           # 'postgresql_psycopg2',
                                 'postgresql', 'mysql',
                                 'sqlite3' or 'oracle'.
```

```
DATABASE_NAME = ''             # Or path to database file
                                 if using sqlite3.
DATABASE_USER = ''             # Not used with sqlite3.
DATABASE_PASSWORD = ''         # Not used with sqlite3.
DATABASE_HOST = ''             # Set to empty string for localhost.
                                 Not used with sqlite3.
DATABASE_PORT = ''             # Set to empty string for default.
                                 Not used with sqlite3.
```

Modifichiamo le prime due righe in questo modo:

```
DATABASE_ENGINE = 'sqlite3'    # 'postgresql_psycopg2',
                                 'postgresql', 'mysql',
                                 'sqlite3' or 'oracle'.
DATABASE_NAME = 'libreria.db'  # Or path to database file
                                 if using sqlite3.
```

Abbiamo così indicato a Django di usare il database engine SQLite e di scrivere i dati nel file `libreria.db`.
L'ultima modifica da fare nel file `settings.py` riguarda la lista delle applicazioni. Cercando la stringa `INSTALLED_APPS`, troviamo questa variabile di tipo tupla:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
)
```

Inseriamo come ultimi due elementi di questa tupla l'applicazione di `admin` e la nostra applicazione `libreria`:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'libreria',
)
```

L'applicazione di `admin` viene usata nella stragrande maggioranza dei progetti, tant'è vero che il prossimo capitolo le è dedicato in toto.
La virgola dopo `'libreria'` non sarebbe necessaria, visto che abbiamo inserito l'ultimo elemento della tupla `INSTALLED_APPS`. Il mio consiglio è comunque quello di inserirla sempre: questa abitudine non fa male e quasi sicuramente vi tornerà utile quando dovrete inserire altre elementi dopo l'ultimo della tupla.

SQLITE

SQLite è un ottimo database che può essere usato non solo per i test ma anche per progetti di dimensioni medie, dove non si devono gestire milioni di record e migliaia di accessi contemporanei. I suoi autori sostengono che sia l'SQL engine più usato al mondo, dato che si trova in moltissime applicazioni su desktop, oltre che su cellulari, palmari, lettori MP3 e altro ancora. Il sorgente di SQLite è di dominio pubblico (`http://sqlite.org`).

Creiamo un modello

Ora che abbiamo tutti gli elementi necessari, possiamo finalmente creare i nostri primi modelli.
Visto che abbiamo creato un'applicazione chiamata `libreria`, definiamo tre classi di oggetti attinenti che, con estrema fantasia, chiameremo Autore, Genere e Libro.
Modifichiamo il file `models.py`, presente nella directory `libreria`, e inseriamo in fondo al file la descrizione dei nostri oggetti (le prime due righe sono già nel file):

```
from django.db import models
# Create your models here.

class Autore(models.Model):
    nome = models.CharField(max_length=50)
    cognome = models.CharField(max_length=50)

class Genere(models.Model):
    descrizione = models.CharField(max_length=30)

class Libro(models.Model):
    titolo = models.CharField(max_length=200)
    autore = models.ForeignKey(Autore)
    genere = models.ForeignKey(Genere)
```

L'oggetto Autore ha due attributi, `nome` e `cognome`, che sono due stringhe lunghe fino a 50 caratteri.
L'oggetto Genere ha un solo attributo, `descrizione`, che è una stringa lunga fino a 30 caratteri.
L'oggetto Libro è quello più interessante: ha un attributo `titolo` che è una stringa lunga 200 caratteri e ha una relazione uno-a-molti verso gli oggetti Genere e Autore.
In altre parole queste due relazioni indicano che un Libro appartiene a un Genere ed è scritto da un Autore. Ovviamente possono esistere più oggetti Libro per un dato Genere e un Autore può scrivere più libri.

> **UN SOLO AUTORE?**
>
> Questa è una limitazione che nella realtà non è certamente accettabile: i libri possono avere più di un autore. Anzi, a volte possono avere più persone con ruoli diversi: autore, curatore, traduttore e via dicendo. Vedremo comunque come modificare un modello nelle portate della seconda parte di questo libro.

Abbiamo così completato la descrizione dei nostri tre modelli e possiamo far generare a Django il database SQLite con il seguente comando:

```
python manage.py syncdb
```

Nella Figura 2.1 vediamo l'output del precedente comando.

> **SUPERUSER**
>
> A un certo punto ci viene chiesto di creare un superutente, indicando un nome, un indirizzo e-mail e una password. Facciamolo pure, ma attenzione: non dimentichiamo cosa abbiamo inserito, perché ci tornerà utile nel prossimo capitolo, dedicato al pannello di amministrazione web generato automaticamente da Django.

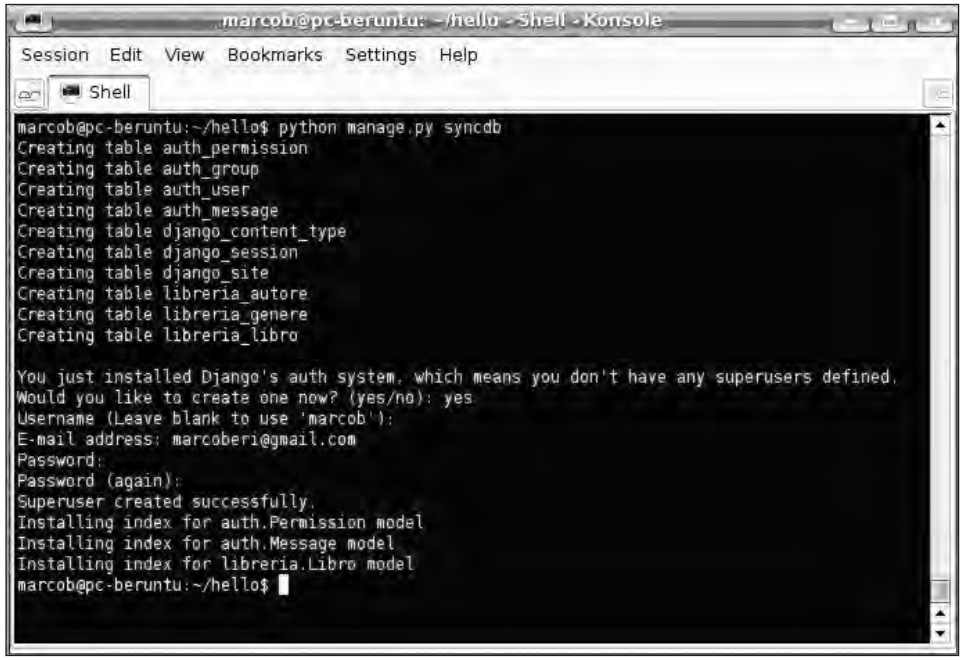

Figura 2.1 Creazione del database libreria.db.

Le tabelle create da Django sono state inserite nel file `libreria.db`, nella directory principale del nostro progetto, grazie alla seguente riga che avevamo modificato nel file `settings.py`:

```
DATABASE_NAME = 'libreria.db'  # Or path to database file
                               if using sqlite3.
```

Parte delle tabelle create è usata da Django stesso (sono quelle i cui nomi iniziano per `django_` e per `auth_`), mentre le nostre tre sono `libreria_autore`, `libreria_genere` e `libreria_libro`. La prima parte del loro nome coincide con il nome dell'applicazione, mentre la seconda parte è il nome che abbiamo dato alle classi di oggetti.
Con questo passo il nostro database è pronto e possiamo finalmente iniziare a inserire i dati.

Inseriamo i dati

Per inserire qualche dato di prova è sufficiente eseguire Python in modalità interattiva, nella directory del progetto `hello`:

```
python manage.py shell
```

Dalla shell interattiva possiamo importare i modelli che abbiamo definito nel file `libreria/models.py` e creare oggetti di tipo Genere:

```
>>> from libreria.models import *
>>> romanzo = Genere(descrizione="romanzo")
>>> romanzo.save()
```

```
>>> fantascienza = Genere(descrizione="fantascienza")
>>> fantascienza.save()
>>> giallo = Genere(descrizione="giallo")
>>> giallo.save()
>>>
```

È il metodo `save` che scatena effettivamente la creazione del record nel database SQLite. Adesso creiamo qualche oggetto di tipo Autore:

```
>>> suskind = Autore(cognome="Suskind", nome="Patrick")
>>> suskind.save()
>>> asimov = Autore(cognome="Asimov", nome="Isaac")
>>> asimov.save()
>>> ellroy = Autore(cognome="Ellroy", nome="James")
>>> ellroy.save()
>>>
```

Infine creiamo i libri della nostra libreria, usando per gli attributi `genere` e `autore` proprio gli oggetti che abbiamo appena creato:

```
>>> profumo = Libro(titolo="Il Profumo",
... genere=romanzo,
... autore=suskind)
>>> profumo.save()
>>>
>>> dalia = Libro(titolo="La dalia nera",
... genere=giallo,
... autore=ellroy)
>>> dalia.save()
>>>
>>> fondazione = Libro(titolo="Fondazione",
... genere=fantascienza,
... autore=asimov)
>>> fondazione.save()
>>>
```

Con queste istruzioni abbiamo creato dei record nelle tabelle del database SQLite. Ma se avessimo scelto come database di backend PostgreSQL, MySQL o Oracle, queste istruzioni non sarebbero cambiate di una virgola. Avremmo semplicemente dovuto cambiare i parametri di configurazione del database nel file `settings.py`.
Possiamo adesso terminare la shell interattiva con l'istruzione `exit()`:

```
>>> exit()
```

Per uscire dalla shell è anche possibile usare il carattere di fine file, che in Windows si ottiene premendo i tasti Ctrl+Z e in GNU/Linux e Mac OSX con i tasti Ctrl+D.

Interroghiamo il database

Dal prompt interattivo possiamo anche, ovviamente, recuperare dal database gli oggetti che abbiamo appena salvato. Avviamo nuovamente la shell e importiamo i modelli:

```
python manage.py shell
>>> from libreria.models import *
```

Proviamo ora ad accedere agli oggetti di tipo Genere:

```
>>> Genere.objects.all()
[<Genere: Genere object>, <Genere: Genere object>, <Genere: Genere object>]
>>>
```

L'attributo `objects` della classe Genere è il cosiddetto *manager* che ci permette di accedere agli oggetti. Il suo metodo `all` li restituisce tutti, infatti la lista stampata contiene proprio tre oggetti.
Proviamo ora a visualizzarli in modo più comprensibile:

```
>>> for genere in Genere.objects.all():
...     print genere.descrizione
romanzo
fantascienza
giallo
>>>
```

In Python un'istanza di una classe viene, per default, rappresentata con questo standard: `<Nome della classe: Nome della classe object>`. Definendo un metodo `__unicode__` della classe possiamo ottenere un formato più comprensibile.
Per esempio, definiamo dinamicamente questo metodo nella classe `Genere`: e vediamo come diventa l'output di `Genere.objects.all()`:

```
>>> def __unicode__(self): return self.descrizione
>>> Genere.__unicode__ = __unicode__
>>> Genere.objects.all()
[<Genere: romanzo>, <Genere: fantascienza>, <Genere: giallo>]
>>>
```

MA IL METODO __UNICODE__ RIMARRÀ?

Il metodo `__unicode__`, che abbiamo definito dinamicamente dalla shell interattiva, sparirà nel momento in cui usciremo da essa. Per rendere il metodo permanente dovremmo aggiungerlo nel sorgente `models.py`. Ma non facciamolo adesso: attendiamo il prossimo capitolo.

Notiamo come l'ordine di visualizzazione equivalga all'ordine di inserimento. Possiamo, se lo desideriamo, ordinare i generi in base all'attributo `descrizione`:

```
>>> for genere in Genere.objects.all().order_by("descrizione"):
...     print genere.descrizione
fantascienza
giallo
romanzo
>>>
```

Ci è bastato *aggiungere* il metodo `order_by` per indicare all'ORM di Django l'attributo con cui desideriamo ordinare gli oggetti.
Nella Figura 2.2 vediamo le interrogazioni svolte sino a questo punto.
Se vogliamo invece estrarre solo una parte degli oggetti, possiamo usare il metodo `filter`:

```
>>> for autore in Autore.objects.filter(nome="James"):
...    print autore.nome, autore.cognome
...
James Ellroy
>>>
```

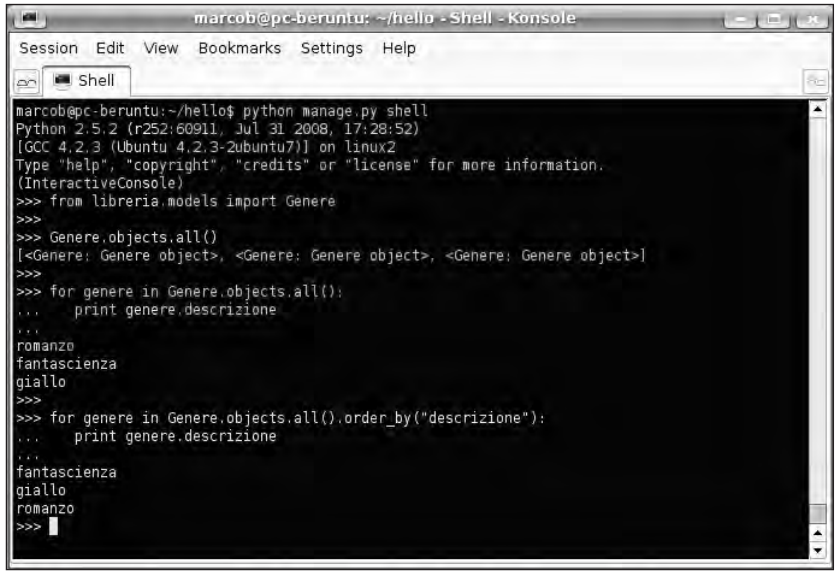

Figura 2.2 Interrogazione del database dalla shell interattiva.

Il metodo `filter` accetta anche dei parametri complessi, come nell'esempio seguente che ci permette di estrarre solo gli autori che contengono la lettera "o" nel cognome:

```
>>> for autore in Autore.objects.filter(cognome__contains="o"):
...     print autore.nome, autore.cognome
...
Isaac Asimov
James Ellroy
>>>
```

In questo caso abbiamo aggiunto al nome dell'attributo `cognome`, la stringa `__contains` (attenzione ai due underscore!).
Proviamo ora qualcosa di un pochino più complesso, cercando solo i libri della nostra libreria scritti da James Ellroy:

```
>>> for libro in Libro.objects.filter(autore__cognome="Ellroy"):
...     print libro.titolo
...
La dalia nera
>>>
```

Questa volta abbiamo usato come filtro l'attributo `cognome` dell'autore del libro, grazie al parametro `autore__cognome` (sempre con due underscore in mezzo).

> **LA PUNTA DELL'ICEBERG**
>
> Quello che abbiamo visto in questo capitolo è solo la punta dell'iceberg dell'ORM di Django. Nella seconda parte del libro vedremo in dettaglio l'enorme varietà di tipi di dati che possiamo usare per definire i nostri oggetti (per esempio, sinora abbiamo visto solo il tipo `CharField`, mentre in tutto esistono circa una trentina di tipi nativi diversi). Vedremo anche come sia possibile usare relazioni *molti-a-molti* o *uno-a-uno* e interrogare in maniera molto complessa il database, collegando in cascata diversi filtri.

I test

Django ha già al suo interno dei test di autodiagnostica. Per eseguirli basta il comando seguente:

```
python manage.py test
```

Il risultato di questo comando eseguito sul nostro progetto dovrebbe essere il seguente:

```
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table libreria_autore
Creating table libreria_genere
Creating table libreria_libro
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for libreria.Libro model
................
----------------------------------------------------------------------
Ran 16 tests in 1.813s

OK
Destroying test database...
```

I 16 test eseguiti comprendo anche la creazione di un database di test che viene, al termine dei test, cancellato.
Possiamo (ma sarebbe meglio dire dobbiamo) aggiungere anche qualche test per il nostro codice. Creiamo un file `tests.py` nella directory `libreria` e aggiungiamo una copia di tutta la sessione di test che abbiamo appena eseguito dalla shell di Python:

```
"""
>>> from libreria.models import *
>>> romanzo = Genere(descrizione="romanzo")
>>> romanzo.save()
>>> fantascienza = Genere(descrizione="fantascienza")
>>> fantascienza.save()
>>> giallo = Genere(descrizione="giallo")
>>> giallo.save()
>>>
>>> suskind = Autore(cognome="Suskind", nome="Patrick")
>>> suskind.save()
>>> asimov = Autore(cognome="Asimov", nome="Isaac")
>>> asimov.save()
>>> ellroy = Autore(cognome="Ellroy", nome="James")
>>> ellroy.save()
>>>
>>> profumo = Libro(titolo="Il Profumo",
... genere=romanzo,
... autore=suskind)
```

```
>>> profumo.save()
>>>
>>> dalia = Libro(titolo="La dalia nera",
... genere=giallo,
... autore=ellroy)
>>> dalia.save()
>>>
>>> fondazione = Libro(titolo="Fondazione",
... genere=fantascienza,
... autore=asimov)
>>> fondazione.save()
>>>
>>> Genere.objects.all()
[<Genere: Genere object>, <Genere: Genere object>, <Genere: Genere object>]
>>>
>>> for genere in Genere.objects.all():
...     print genere.descrizione
romanzo
fantascienza
giallo
>>> def __unicode__(self): return self.descrizione
>>> Genere.__unicode__ = __unicode__
>>> Genere.objects.all()
[<Genere: romanzo>, <Genere: fantascienza>, <Genere: giallo>]
>>>
>>> for genere in Genere.objects.all().order_by(“descrizione”):
...     print genere.descrizione
fantascienza
giallo
romanzo
>>>
>>> for autore in Autore.objects.filter(nome="James"):
...    print autore.nome, autore.cognome
...
James Ellroy
>>>
"""
```

Rieseguiamo i test con lo stesso comando:

```
python manage.py test
```

Questa volta il risultato dovrebbe essere lievemente diverso:

```
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table libreria_autore
Creating table libreria_genere
Creating table libreria_libro
Installing index for auth.Permission model
Installing index for auth.Message model
```

```
Installing index for admin.LogEntry model
Installing index for libreria.Libro model
.................
----------------------------------------------------------------------
Ran 17 tests in 1.672s

OK
Destroying test database...
```

La differenza sta in un puntino in più e nel totale che da 16 è passato a 17 test eseguiti correttamente.
Proviamo ora a introdurre di proposito un errore nel nostro `models.py`, per vedere cosa succede con i test.
Modifichiamo il campo `autore` come nelle due seguenti righe in grassetto (manteniamo la riga originale commentata, in modo da poter poi tornare con facilità alla situazione corretta):

```
class Libro(models.Model):
    titolo = models.CharField(max_length=200)
    #autore = models.ForeignKey(Autore)
    autore = models.CharField(max_length=200)
    genere = models.ForeignKey(Genere)
```

In questo modo `autore` non sarà più collegato alla classe `Autore`. Proviamo a eseguire nuovamente i test. Ecco il risultato:

```
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table libreria_autore
Creating table libreria_genere
Creating table libreria_libro
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for libreria.Libro model
.............F
======================================================================
FAIL: Doctest: libreria.tests
----------------------------------------------------------------------
Traceback (most recent call last):
```

[Qui in mezzo ci sarà una serie piuttosto lunga di messaggi di errore.]

```
        raise FieldError("Join on field %r not permitted." % name)
    FieldError: Join on field 'autore' not permitted.
----------------------------------------------------------------------
Ran 17 tests in 1.891s
FAILED (failures=1)
Destroying test database...
```

La segnalazione che ci interessa è l'ultima:

```
FieldError: Join on field 'autore' not permitted.
```

Non è più possibile, a causa della modifica che abbiamo fatto, eseguire il join tra `Libro` e `Autore`.
A questo punto, in una situazione reale, dovremmo capire se l'errore è stato indotto da un baco o, situazione più inusuale, da una modifica del codice che ha invalidato un test. Nel nostro caso abbiamo introdotto di proposito un baco, per cui risistemiamo il campo `autore` in `models.py` e rieseguiamo i test per accertarci che tutto sia a posto:

```
----------------------------------------------------------------------
Ran 17 tests in 1.672s
OK

Destroying test database...
```

Riepilogo

In questo capitolo abbiamo visto:

- come creare un'applicazione all'interno di un progetto (con il comando `manage.py startapp <nome applicazione>`);
- come modificare alcuni parametri del progetto per definire il database da usare e le applicazioni da caricare (nel file `settings.py`);
- come creare un modello (nel file `models.py`);
- come creare il database e il superuser (con il comando `manage.py syncdb`);
- come inserire dei dati da shell interattiva (con il comando `manage.py shell`);
- come interrogare il database da shell interattiva;
- come eseguire dei test (con il comando `manage.py test`);
- infine, come eseguire dei test scritti da noi.

Capitolo 3

L'Admin

In questo capitolo

- **L'applicazione Admin**
- **Il pannello di Admin**
- **Gestione dei dati nell'Admin**
- **I test**
- **AdminDocs**

"Se non ci mostrate il Graal, attaccheremo il castello con la forza."
I Monty Python e il Sacro Graal

Come sarebbe bello disegnare il database e, fatto questo, avere già pronti i form per inserire via browser i dati, non è vero?
E magari riuscire così a tranquillizzare il cliente (che probabilmente da giorni ci tormenta per "vedere qualcosa di pronto").
Bene, con Django questo è possibile.
Se per convincere uno scettico dovessi scegliere la singola caratteristica di Django che ne rappresenti il valore, la versatilità e la facilità di sviluppo, opterei proprio per l'interfaccia di Admin autogenerata.

L'applicazione Admin

Django contiene una serie di applicazioni, già pronte per l'uso, e una di queste si chiama *admin*. Per usarla è necessario inserire una riga nel file `settings.py`, proprio come abbiamo già fatto nel capitolo precedente:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'libreria',
)
```

Il prossimo file da modificare è `urls.py` e i cambiamenti sono minimi, visto che non dobbiamo far altro che "decommentare" le tre righe indicate in grassetto:

```
# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()
...
    # Uncomment the next line to enable the admin:
    (r'^admin/(.*)', admin.site.root),

    (r'^hello/$', "views.hello"),
)
```

L'ultimo file, da creare *ex-novo*, è `admin.py` che deve stare nella directory `libreria` assieme a `models.py`. Il codice da inserire è il seguente:

```
from django.contrib import admin
from libreria.models import *

admin.site.register(Autore)
admin.site.register(Genere)
admin.site.register(Libro)
```

La struttura delle directory e dei file del progetto `hello` dovrebbe essere a questo punto la seguente:

```
__init__.py
libreria.db
manage.py
settings.py
urls.py
views.py
libreria\__init__.py
libreria\admin.py
libreria\models.py
libreria\views.py
```

Avviamo, come al solito, il server integrato in Django:

```
python manage.py runserver
```

Apriamo con il browser l'indirizzo `http://127.0.0.1:8000/admin`. Se non abbiamo commesso errori, dovremmo vedere la stessa immagine mostrata nella Figura 3.1. Adesso, se non siamo dei novelli Golfera, prendiamo il foglietto dove abbiamo segnato il nome e la password del superuser, creato nel capitolo precedente, e inseriamoli nel form di login.

> **NON ME LI SONO SEGNATI E ORA NON ME LI RICORDO PIÙ...**
>
> Lo confesso: è successo anche a me esattamente in questo momento. Ma niente paura, per ricreare il superuser ci basta eseguire il seguente comando:
>
> `python manage.py createsuperuser`
>
> Per la cronaca, Gianni Golfera è uno mnemonista in grado di memorizzare, parola per parola, 261 libri "prevalentemente di filosofia ermetica" (da `http://it.wikipedia.org/wiki/Gianni_Golfera`).

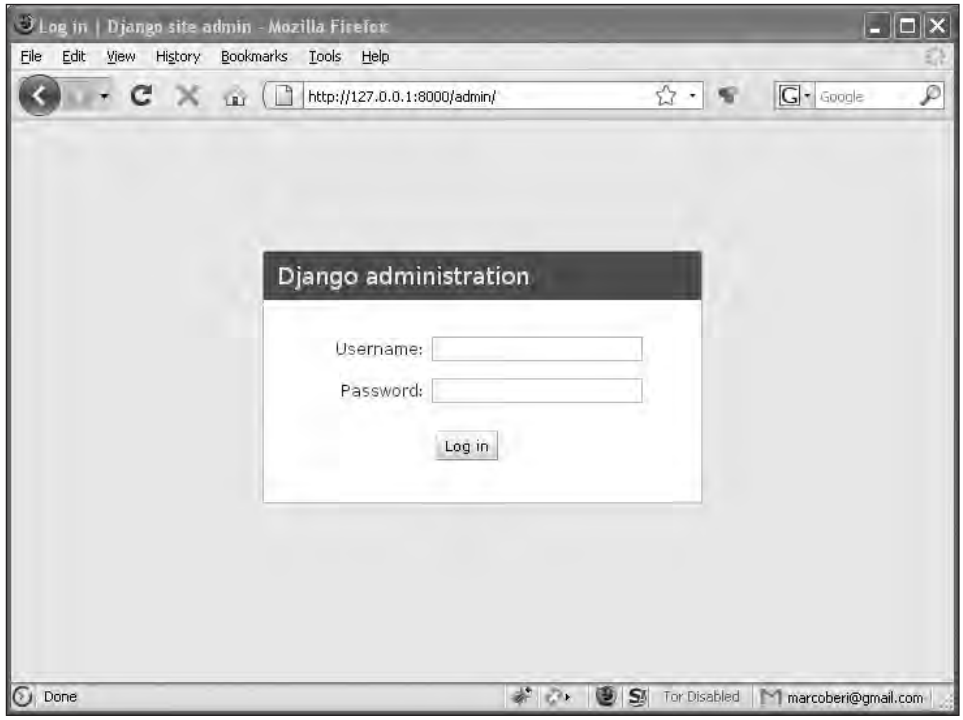

Figura 3.1 Il login del pannello di amministrazione.

Dopo aver inserito username e password, facciamo clic sul pulsante *Log in* e vedremo finalmente, come nella Figura 3.2, il pannello di amministrazione vero e proprio.

Figura 3.2 Il pannello di amministrazione.

Il pannello di Admin

Il pannello di amministrazione che abbiamo di fronte mostra una serie di messaggi informativi: il nome della persona collegata in alto a destra (*Welcome, marcob*), subito sotto l'elenco delle ultime operazioni svolte (*Recent Actions*). Sotto la scritta *Site administration* troviamo poi gli insiemi di oggetti che possiamo gestire raggruppati per applicazione.

> **AUTH E SITES**
>
> Due applicazioni sono presenti in automatico: Auth e Sites. Vedremo più avanti in dettaglio il loro funzionamento. Per ora ci basti sapere che la prima, Auth, si occupa della gestione della sicurezza (utenti, gruppi e permessi), mentre la seconda, Sites, permette di differenziare tra loro siti diversi gestiti con una sola installazione di Django.

Sotto la nostra applicazione, Libreria, vediamo uno strano elenco: *Autores*, *Generes* e *Libros*. Non è che Django si sia messo improvvisamente a parlare in spagnolo, anzi, continua imperterrito a parlare in inglese e aggiunge la "s" per cercare di ottenere i plurali delle parole (come in *Groups*, *Users*, *Actions* e via dicendo).
Come primo passo indichiamo a Django che vogliamo che parli in italiano.
Interrompiamo il server, apriamo in modifica il file di configurazione `settings.py`, cerchiamo la seguente riga:

```
LANGUAGE_CODE = 'en-us'
```

e modifichiamola così:

```
LANGUAGE_CODE = 'it-IT'
```

Riavviamo il server e ricarichiamo la pagina del browser. Dovremmo ora vedere la pagina come appare nella Figura 3.3.
Le cose vanno meglio, quasi tutto è in italiano ma abbiamo sempre *Autores*, *Generes* e *Libros*. Django non può conoscere il plurale dei nostri oggetti Autore, Genere e Libro, siamo noi a doverglielo spiegare. Per farlo modifichiamo il file `models.py` nella directory `libreria` del nostro progetto e a ogni modello aggiungiamo le seguenti due righe:

```
    class Meta:
        verbose_name_plural = "Autori"
```

Ovviamente al posto di "`Autori`" inseriamo il valore corretto per ogni modello. Il file `models.py`, alla fine delle modifiche, dovrebbe essere questo:

```
from django.db import models
# Create your models here.

class Autore(models.Model):
    nome = models.CharField(max_length=50)
    cognome = models.CharField(max_length=50)
    class Meta:
        verbose_name_plural = "Autori"

class Genere(models.Model):
    descrizione = models.CharField(max_length=30)
    class Meta:
        verbose_name_plural = "Generi"
```

Figura 3.3 Il pannello di amministrazione quasi in italiano.

```
class Libro(models.Model):
    titolo = models.CharField(max_length=200)
    autore = models.ForeignKey(Autore)
    genere = models.ForeignKey(Genere)
    class Meta:
        verbose_name_plural = "Libri"
```

Ricarichiamo la pagina nel browser; ora dovremmo vedere un pannello simile a quello nella Figura 3.4.

Figura 3.4 Il pannello di amministrazione in italiano.

STOP-AND-GO A GO-GO DEL SERVER INTEGRATO? NON SERVE!

In realtà non è necessario interrompere e riavviare il server prima di ogni modifica del codice. Possiamo modificare e salvare il file `models.py`, aprendolo per esempio da una nuova shell, e Django si accorgerà automaticamente dei cambiamenti, ricaricando il modulo python aggiornato. Nella Figura 3.5 vediamo proprio questa situazione.

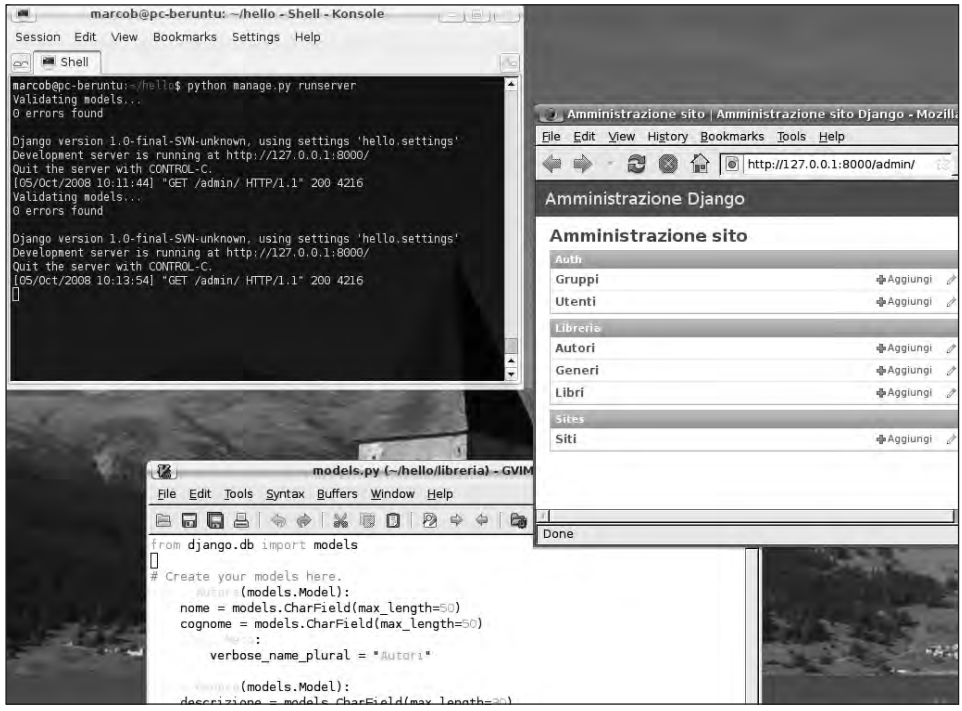

Figura 3.5 Sullo sfondo il server che ricarica autonomamente i sorgenti modificati.

Gestione dei dati nell'Admin

Proviamo ora a entrare nella schermata di gestione di uno dei nostri modelli, per esempio gli Autori. Per farlo è sufficiente fare clic su *Autori* o su *Modifica* (come è facile immaginare, facendo clic su *Aggiungi*, accederemmo direttamente alla funzione di inserimento di un nuovo autore).
La schermata che ci si presenta davanti è un po' diversa, come possiamo vedere nella Figura 3.6. Abbiamo in alto a sinistra i *breadcrumb* (briciole di pane) che indicano che siamo nella gestione degli *Autori* che fanno parte della applicazione *Libreria* che sta nella *Pagina iniziale* dell'Admin. Vediamo a destra un pulsante *Aggiungi autore* (lascio come esercizio al lettore capire a cosa serve) e una lista di tre elementi chiamati tutti *Autore object*.

BREADCRUMB

Il termine *breadcrumb*, letteralmente "briciole di pane", trova origine nella fiaba di Hansel e Gretel, dove i due fratellini usano proprio delle briciole di pane per segnare e poi ritrovare la

Figura 3.6 La gestione degli autori.

strada per tornare a casa. In un sito i breadcrumb, quasi sempre posizionati in alto a sinistra, indicano al navigatore la strada percorsa dalla home page fino alla pagina corrente.

Sicuramente una lista come questa non ci dà sufficienti indicazioni per capire quale autore corrisponde a ciascun elemento. Per risolvere questo problema dobbiamo fare una piccola modifica al file `models.py`, aggiungendo alla classe `Autore` il metodo `__unicode__`:

```
class Autore(models.Model):
    nome = models.CharField(max_length=50)
    cognome = models.CharField(max_length=50)
    def __unicode__(self):
        return u"%s %s" % (self.nome, self.cognome)
    class Meta:
        verbose_name_plural = "Autori"
```

Ricarichiamo la pagina del browser per vedere la lista, molto più comprensibile, mostrata nella Figura 3.7.

Figura 3.7 La gestione (meno criptica) degli autori.

Vista l'utilità del metodo `__unicode__` applichiamo questa modifica anche alle classi `Genere` e `Libro`:

```
class Genere(models.Model):
    descrizione = models.CharField(max_length=30)
    def __unicode__(self):
        return self.descrizione
    class Meta:
        verbose_name_plural = "Generi"

class Libro(models.Model):
    titolo = models.CharField(max_length=200)
    autore = models.ForeignKey(Autore)
    genere = models.ForeignKey(Genere)
    def __unicode__(self):
        return self.titolo
    class Meta:
        verbose_name_plural = "Libri"
```

Facciamo clic su *Patrick Suskind* per modificare il corrispondente autore (Figura 3.8), correggiamo il cognome in *Süskind* e salviamo la modifica facendo clic su *Salva*.

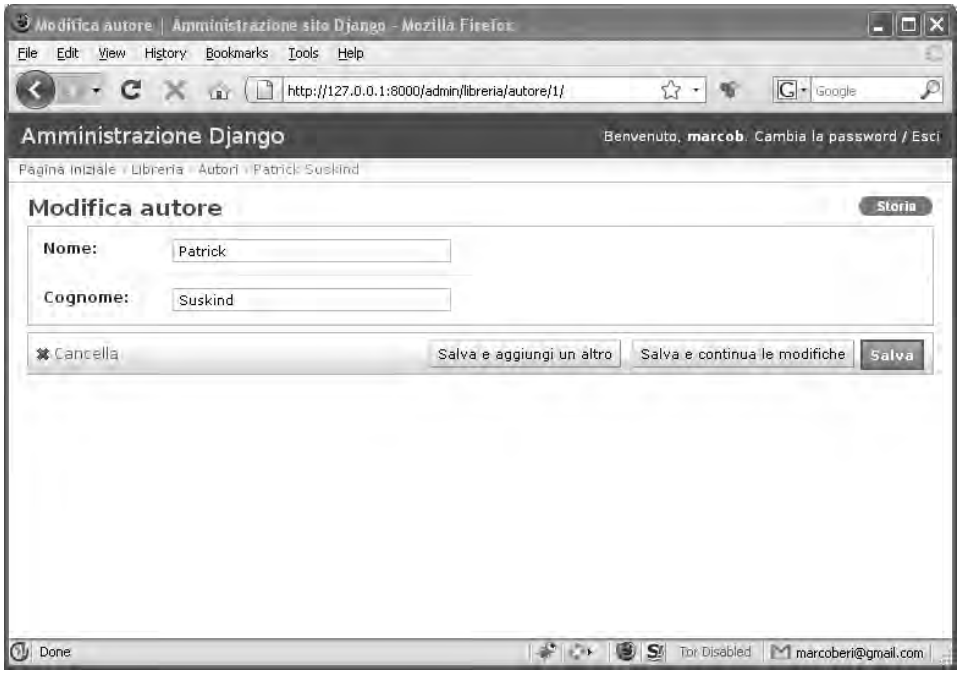

Figura 3.8 Modifica di un autore.

MA COME FACCIO A DIGITARE LA Ü?

Ci sono tre modi differenti per digitare la u con la dieresi: 1) se ho a disposizione il tasto compose (per esempio in GNU/Linux), premo in sequenza compose, la u e infine le doppie virgolette; 2) se sto usando Windows, la sequenza magica è ALT+0252 (le cifre vanno

premute sul tastierino numerico); 3) se non ci riesco con uno dei due metodi precedenti, cerco "Il profumo" in Wikipedia e copio e incollo la ü dal nome dell'autore. Il metodo 3) è normalmente il più usato.

Come vediamo a metà nella Figura 3.9, un messaggio su sfondo giallo ci indica la buona riuscita della nostra operazione.

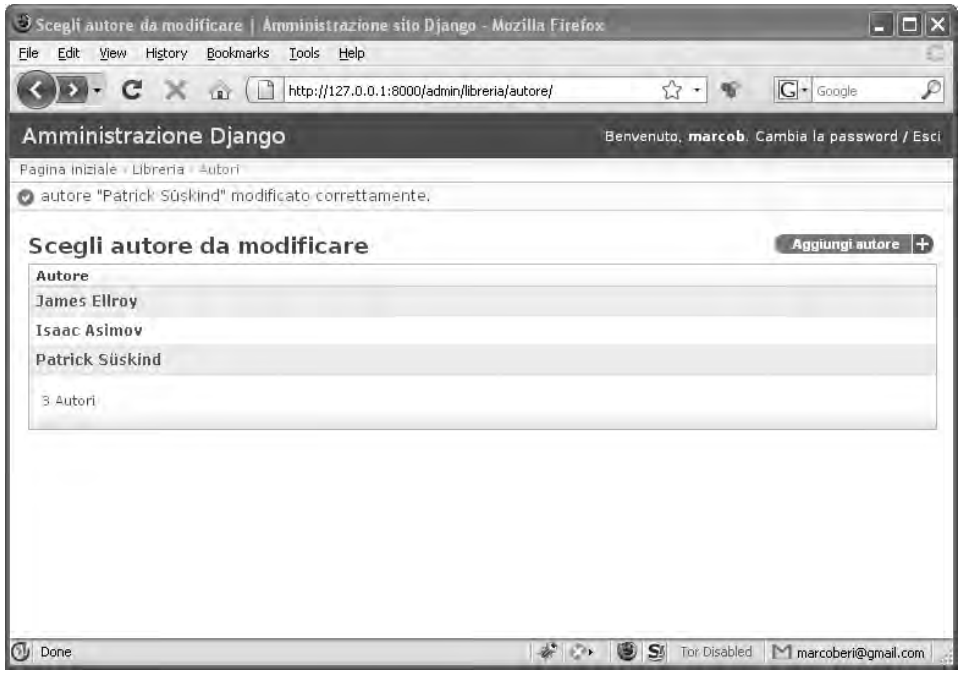

Figura 3.9 Un autore modificato.

Tornando sulla pagina principale dell'Admin (ci basta fare clic sulla prima briciola dei *breadcrumb*), vediamo che qualcosa è cambiato: nel box *Azioni Recenti*, che nella Figura 3.4 non conteneva nulla, ora è apparso un elemento che indica che abbiamo modificato l'autore *Patrick Süskind* (l'icona della matita indica l'operazione di modifica).
Come ultima operazione proviamo a inserire un oggetto Libro, e per farlo facciamo clic direttamente sulla voce *Aggiungi*, quella con un + verde vicino, a destra della voce *Libri*.
Il modulo per inserire un libro è un pochino più complesso: abbiamo un titolo e poi abbiamo due caselle combinate corrispondenti all'autore e al genere del libro (Figura 3.10), contenenti rispettivamente i tre autori e i tre generi presenti nel nostro database.
Il libro che vogliamo aggiungere è *Sei pezzi facili*, quindi inseriamo questo titolo nel corrispondente campo. L'autore è Richard Feynman e il libro è un *saggio*, per cui non abbiamo l'autore giusto e nemmeno il genere. Cosa facciamo? Dobbiamo forse uscire da questa schermata perdendo il titolo già inserito, tornare alla pagina iniziale e inserire autore e genere? Certamente no: facciamo clic sul + verde a destra della casella combinata contenente gli autori e, grazie alla finestra popup che si aprirà, inseriamo l'autore

Figura 3.10 Inserimento di un libro.

Richard Feynman. Dopo aver fatto clic sul pulsante *Salva*, la finestra si chiuderà e nella casella combinata degli autori avremo il nostro nuovo autore già selezionato (Figura 3.11). Facciamo la stessa operazione per il campo seguente, inserendo il genere *saggio*, e infine salviamo il libro così completato. Il messaggio "*libro "Sei pezzi facili" è stato aggiunto correttamente*" ci indica che abbiamo fatto tutto bene.

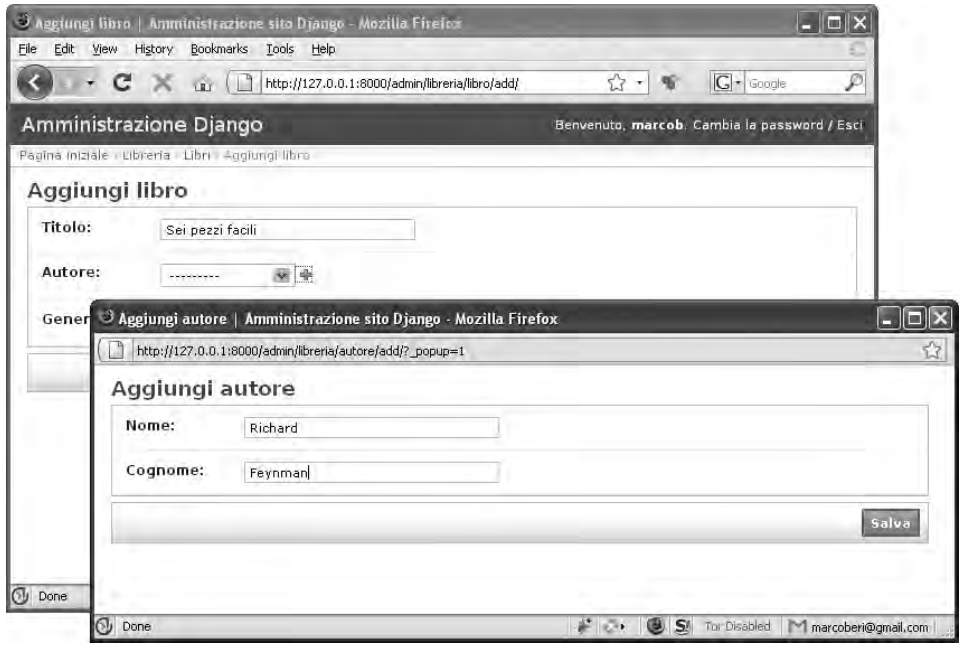

Figura 3.11 Inserimento contestuale di un autore.

Tornando sulla pagina iniziale vediamo che le azioni recenti adesso contengono quattro elementi. Il primo in alto è il più recente (inserimento del libro), quindi subito sotto appaiono l'inserimento del genere e dell'autore. Infine c'è la prima modifica dell'autore del libro *Il profumo* (Figura 3.12).

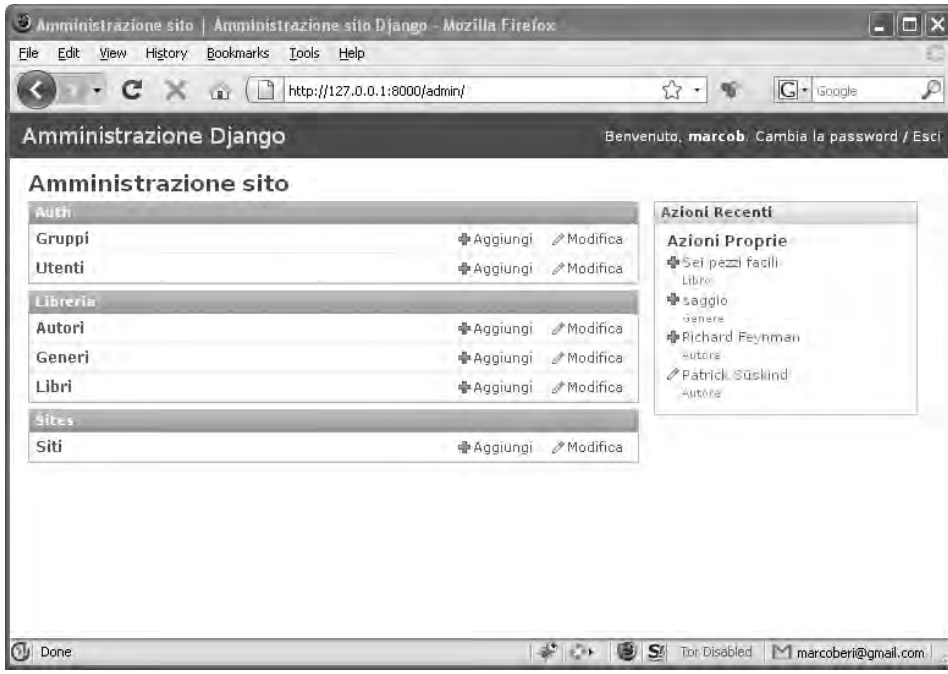

Figura 3.12 Le Azioni Recenti proliferano.

I test

Il nostro codice in questo capitolo è cambiato abbastanza, per cui è una buona cosa provare a rieseguire i test di Django con il comando `manage.py test`:

```
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table libreria_autore
Creating table libreria_genere
Creating table libreria_libro
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for libreria.Libro model
```

```
.............F
======================================================================
FAIL: Doctest: libreria.tests
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python25\lib\site-packages\django\test\_doctest.py",
    line 2180, in runTest
    raise
    self.failureException(self.format_failure(new.getvalue()))
AssertionError: Failed doctest test for libreria.tests
  File "C:\Work\hello\libreria\tests.py", line 0, in tests
----------------------------------------------------------------------
File "C:\Work\hello\libreria\tests.py", line 33, in libreria.tests
Failed example:
    Genere.objects.all()
Expected:
    [<Genere: Genere object>, <Genere: Genere object>, <Genere: Genere object>]
Got:
    [<Genere: romanzo>, <Genere: fantascienza>, <Genere: giallo>]
----------------------------------------------------------------------
Ran 17 tests in 1.735s

FAILED (failures=1)
Destroying test database...
```

Il fallimento dei test è stato introdotto per via dell'aggiunta del metodo `__unicode__` nella classa `Genere`.
Per adeguare i test a questa nuova situazione ci basta, semplicemente, cancellare al punto giusto in `tests.py` il metodo di troppo.
Cerchiamo le seguenti righe:

```
>>> fondazione.save()
>>>
>>> Genere.objects.all()
```

Inseriamo tra loro la riga in grassetto:

```
>>> fondazione.save()
>>>
>>> del Genere.__unicode__
>>>
>>> Genere.objects.all()
```

Rieseguiamo i test e verifichiamo che non ci siano più errori:

```
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table libreria_autore
Creating table libreria_genere
Creating table libreria_libro
Installing index for auth.Permission model
```

```
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for libreria.Libro model
.................
----------------------------------------------------------------------
Ran 17 tests in 1.812s
OK
Destroying test database...
```

Ora i nostri test sono di nuovo aggiornati e corretti rispetto al codice.

> **TDD: TEST DRIVEN DEVELOPMENT (SVILUPPO GUIDATO DAI TEST)**
>
> Esiste una scuola di pensiero che sostiene non si debba mai scrivere del codice di produzione prima di aver scritto i relativi test. Le regole d'oro del TDD sono: 1) Non scrivere codice di produzione a meno che non sia per far passare un test che fallisce; 2) Non scrivere nessun ulteriore codice di test quando quello già scritto è sufficiente per generare un errore; 3) Non scrivere nessun ulteriore codice produzione quando quello già scritto è sufficiente a risolvere un test che fallisce.

AdminDocs

Nel file `urls.py`, nelle vicinanze della riga che abbiamo decommentato per attivare l'Admin, ci sono le seguenti istruzioni:

```
# Uncomment the admin/doc line below and add
                                'django.contrib.admindocs'
# to INSTALLED_APPS to enable admin documentation:
# (r'^admin/doc/', include('django.contrib.admindocs.urls'))
```

Seguiamo il consiglio del commento. Prima di tutto decommentiamo la seguente riga di codice in `urls.py`:

```
# Uncomment the admin/doc line below and add
                                'django.contrib.admindocs'
# to INSTALLED_APPS to enable admin documentation:
(r'^admin/doc/', include('django.contrib.admindocs.urls'))
```

Quindi aggiungiamo `'django.contrib.admindocs'` tra le applicazioni installate in `settings.py`:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.admindocs',
    'libreria',
)
```

Adesso proviamo ad aprire il pannello di amministrazione. Se guardiamo in alto a destra nella Figura 3.13, noteremo che c'è ora un nuovo link *Documentazione*. Usandolo accederemo alla documentazione dell'Admin.
Potrebbe però succedere che, entrando nella *Documentazione*, ci appaia la schermata della Figura 3.14. In questo caso dobbiamo installare l'utilissima libreria `docutils`. Possiamo scaricare l'ultima versione dall'indirizzo `http://docutils.sourceforge.net/`.

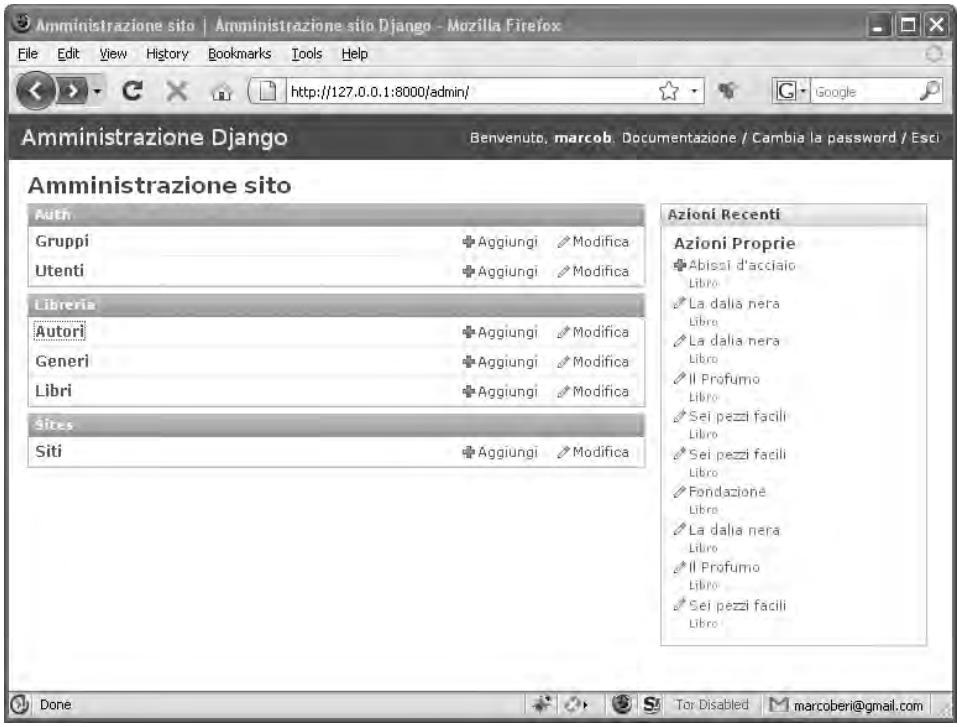

Figura 3.13 Il link Documentazione è apparso in alto.

Figura 3.14 Ci manca docutils!

Il file da installare probabilmente avrà estensione `.tgz`. Per chi usa GNU/Linux o Mac OSX questo non costituisce un problema, ma per chi usa Windows forse sì. Nessuna paura: esiste una utility gratuita che è in grado di leggere archivi `.tgz`. Si chiama 7Zip e la troviamo all'indirizzo `http://www.7zip.org`.
Una volta che abbiamo decompresso il contenuto dell'archivio in una directory temporanea, possiamo installare `docutils` con il seguente comando:

```
python setup.py install
```

A questo punto possiamo riprovare il link *Documentazione* e finalmente accedere alla documentazione dell'Admin.
Nella Figura 3.15 è mostrata la schermata principale dell'AdminDocs da cui possiamo accedere alla documentazione disponibile (Figura 3.16).

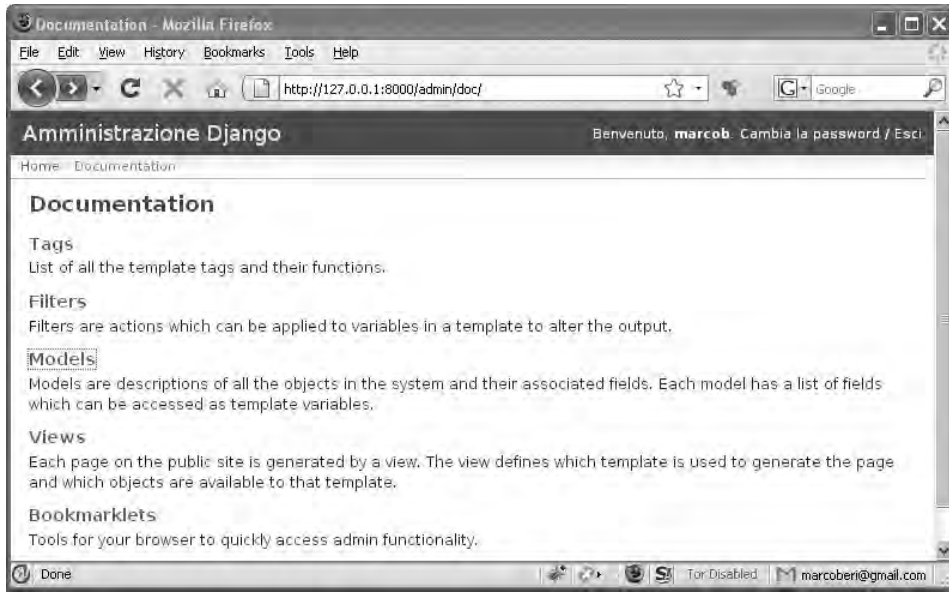

Figura 3.15 Habemus AdminDocs.

Riepilogo

In questo capitolo abbiamo visto:

- come usare l'applicazione Admin (inserendo `django.contrib.admin` nei `settings.py`);
- come accedere al pannello di Admin (`http://url_del_sito/admin`);
- come ricreare il superuser (`python manage.py createsuperuser`);
- come modificare un modello indicando la sua denominazione plurale (con `class Meta: verbose_name_plural = "<Plurale>"`);
- come personalizzare la rappresentazione di un oggetto (con il metodo `__unicode__`);
- come inserire e modificare i dati dal pannello di Admin;

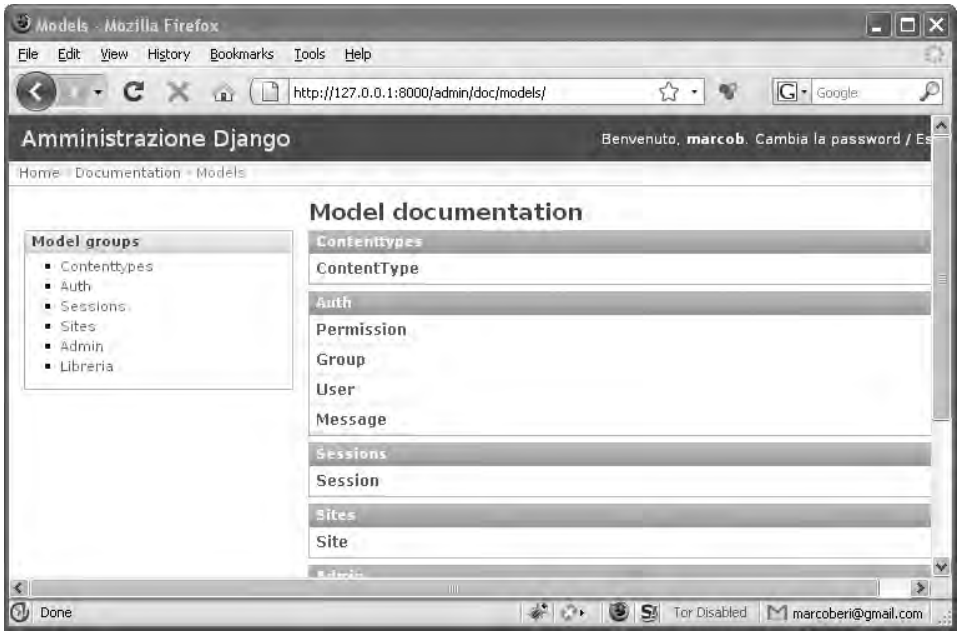

Figura 3.16 L'elenco dei modelli delle applicazioni disponibili.

- come usare in cascata la funzione di inserimento di oggetti collegati tra loro;
- come rieseguire i test per controllare eventuali regressioni nel codice;
- infine, come installare e visualizzare la documentazione online di Admin.

Capitolo 4

Gli URL

In questo capitolo

- **URLconf**
- **URL con parametri**
- **Parametri con nome**

"Nessuno si aspetta l'Inquisizione Spagnola!"
Monty Python's Flying Circus

Quando incontriamo un URL che termina con un valore a scelta tra `.php`, `.asp`, `.aspx`, `.do`, `.jsp`, `.html`, `.htm` e via dicendo, a seconda delle nostre inclinazioni verso la tecnologia che intuiamo dietro a una di queste terminazioni, potremmo già dare un giudizio più o meno negativo sul sito che stiamo visitando.
Oltre a questo rischio, un URL non *pulito* molto probabilmente in futuro non rispetterà uno dei dettami principali dell'inventore del Web, Tim Berners-Lee: "Gli URL fatti bene non cambiano mai".
Per esempio, un link che abbiamo visto nel Capitolo 1 a proposito dell'articolo sul genere delle sigle straniere era:

```
http://www.accademiadellacrusca.it/faq/faq_risp.
php?id=5469&ctg_id=93
```

È praticamente impossibile che questo link non venga modificato in un futuro più o meno prossimo, e questo causerà un errore ai posteri che leggeranno questo paragrafo (quindi, tu, lettore del futuro, non prendertela con il povero autore).
Un URL molto più semplice da ricordare e facile da mantenere avrebbe potuto essere:

```
http://www.accademiadellacrusca.it/faq/articolo_
sigle_straniere
```

Magari questo link funzionerà per il suddetto lettore, perché il sito dell'Accademia della Crusca avrà nel frattempo adottato Django.
Come vedremo in questo capitolo, Django sposa in pieno questa filosofia perché prevede un livello di

mappatura tra gli URL richiesti dal browser e le viste, le funzioni che servono i dati, permettendo quindi una pulizia e una eleganza insuperabili nella definizione dello schema degli URL dei nostri siti.

> **URL/URI/URN**
>
> In realtà Tim Berners-Lee si riferiva innanzitutto agli URI (*Uniform Resource Identifier*) e solo in un secondo momento agli URL e agli URN (*Uniform Resource Name*). Oramai la differenza tra questi termini sta sparendo, a favore del termine "URL", per cui possiamo usare, per nostra comodità, sempre quest'ultimo acronimo. In fondo, un acronimo al giorno toglie il medico di torno (dovevo trovare una scusa per avervene propinati quattro in un solo paragrafo).

URLconf

La mappatura tra gli URL di un'applicazione e le relative viste, le funzioni che servono i dati, avviene attraverso un modulo Python solitamente chiamato URLconf.
Nel nostro progetto `hello` abbiamo già un modulo URLconf, il file `urls.py`:

```
from django.conf.urls.defaults import *

    # Uncomment the next two lines to enable the admin:
    from django.contrib import admin
    admin.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^hello/', include('hello.foo.urls')),

    # Uncomment the admin/doc line below and add
    # 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    (r'^admin/(.*)', admin.site.root),

    (r'^hello/$', "views.hello"),
)
```

La parte che ci interessa in questo momento è l'oggetto `urlpatterns` creato con la funzione `patterns`. Gli argomenti passati alla funzione (a parte il primo "") sono una lista di tuple composte da una espressione regolare e da una stringa o una funzione.
Quando, attraverso un browser, un utente chiede una pagina, Django scorre la lista fino a trovare una espressione che soddisfa l'URL richiesto, e chiama la corrispondente funzione.
Naturalmente dall'URL viene rimosso il nome del sito, per cui `http://127.0.0.1:8000/admin/` diventa solo `admin/` durante la ricerca della corrispondenza.
Vediamo la tupla che abbiamo inserito in fondo al file nel Capitolo 1:

```
(r'^hello/$', "views.hello"),
```

L'espressione regolare `^hello/$` indica l'URL `hello/` in corrispondenza del quale viene chiamata la funzione `views.hello`.
Avviamo, nel caso non sia ancora in esecuzione dal Capitolo 3, il server di Django (oramai sappiamo come si fa: `python manage.py runserver`) e con il browser apriamo l'indirizzo `http://127.0.0.1:8000/hello/`. La scritta "Hello world" ci garantisce che tutto è a posto.
Proviamo ora ad aprire l'indirizzo `http://127.0.0.1:8000/libri`. La schermata che appare, simile a quella nella Figura 4.1, è la schermata di debug di Django. In questo caso l'errore è *404 Page not found* (pagina non trovata).
Questa pagina ci appare perché nei `settings.py` abbiamo la variabile `DEBUG` valorizzata a `True`:

```
# Django settings for hello project.
DEBUG = True
```

Figura 4.1 Un URL inesistente.

> **DEBUG**
>
> È consigliabile tenere `Debug=True` durante lo sviluppo di un sito perché le informazioni della pagina di debug sono utilissime, quasi insostituibili, per correggere gli errori. Per esempio in questo caso, se avessimo `Debug=False`, riceveremmo un generico errore 500, senza nessun'altra indicazione, perché non abbiamo definito un template apposito per visualizzare gli errori (l'eccezione è di tipo `TemplateDoesNotExist`).

Django ci dice che ha ricevuto una richiesta di tipo `GET`, che sta usando l'URLconf definito in `urls.py` e che l'URL corrente `libri` non corrisponde a nessun elemento.

Proviamo allora a inserire un nuovo valore in fondo alla lista in `urls.py`:

```
(r'^libri/$', "libreria.views.libri"),
```

Modifichiamo quindi il file `libreria/views.py`, inserendo il codice seguente, senza preoccuparci dell'HTML *mischiato* al codice python (pratica assolutamente da evitare, ma ancora non sappiamo cosa sono i *template*):

```
# Create your views here.
from django.http import HttpResponse
from models import *

def libri(request):
    elenco = ""
    for libro in Libro.objects.all().order_by('titolo'):
        elenco += "'%s' di %s, %s<br>" % (libro.titolo,
                                          libro.autore, libro.genere)

    return HttpResponse(elenco)
```

> **COS'È IL PARAMETRO REQUEST?**
>
> Per adesso ci basta sapere che Django passa sempre alle viste, come primo parametro, una variabile di tipo `HttpRequest` che può essere usata per reperire diverse informazioni sul tipo di richiesta e sui dati inviati dal browser.

Ricarichiamo la pagina `http://127.0.0.1:8000/libri` ed ecco, come nella Figura 4.2, la lista dei nostri libri con i corrispondenti autori e generi.

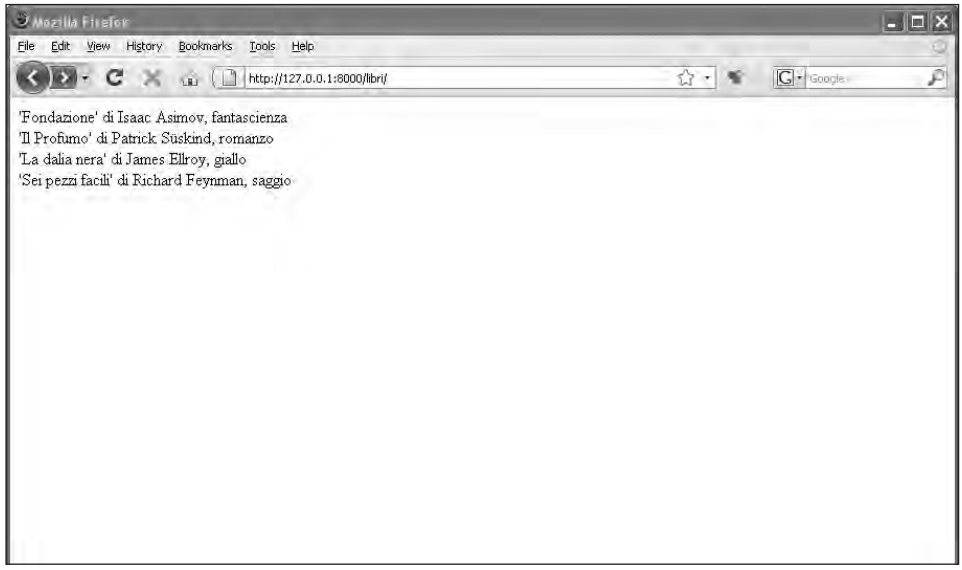

Figura 4.2 Una vista esistente.

URL con parametri

Ora che abbiamo creato un URL per ottenere la lista di libri, vogliamo accedere a un singolo elemento. Come possiamo fare? Non possiamo certo creare un URL per ogni libro.
La soluzione è quella di usare delle espressioni regolari con delle parti variabili. Inseriamo questa nuova tupla in `urls.py`:

```
(r'^libri/(\d+)/$', "libreria.views.libro"),
```

> **ESPRESSIONI REGOLARI: I QUANTIFICATORI**
>
> L'espressione regolare `'^libri/(\d+)/$'` è soddisfatta dalla stringa `/libri/` seguita da uno o più caratteri numerici e quindi dal carattere `/`. I caratteri numerici in una espressione regolare sono indicati da `\d`, mentre il quantificatore "uno o più" è indicato dal `+`. Altri quantificatori sono `*` (zero o più) e `?` (zero o uno).

Definiamo ora questa funzione in fondo a `libreria/views.py`:

```
def libro(request, id):
    try:
        libro = Libro.objects.get(pk=id)
        return HttpResponse("'%s' di %s, %s<br>" % (libro.titolo,
                                    libro.autore, libro.genere))
    except Libro.DoesNotExist:
        return HttpResponse("Codice %s inesistente" % id)
```

> **COS'È GET(PK=ID)?**
>
> L'ORM di Django, dietro le quinte, genera una chiave primaria nel caso non l'abbiamo definita noi manualmente. Nella classe `Libro`, come nelle altre due classi `Genere` e `Autore`, non l'abbiamo fatto, quindi Django ha creato un campo univoco `id`, che assume un valore crescente per ogni record inserito. Il metodo `.get` del manager `.objects`, serve a leggere un record singolo in base alla condizione passata. Nel nostro caso la condizione `pk=id` ci permette di recuperare il libro con la chiave primaria uguale al codice passato nell'URL.

Questa funzione verrà chiamata da Django solo e soltanto in corrispondenza di un URL nel formato previsto. Proviamo ad aprire con il browser i seguenti link:

- `http://127.0.0.1:8000/libri/1/`
- `http://127.0.0.1:8000/libri/99/`
- `http://127.0.0.1:8000/libri/a/`

Cosa succede? Nella Figura 4.3 troviamo la risposta. Nei primi due casi viene chiamata la nostra funzione `libro`, mentre nel terzo caso, visto che il parametro `a` non è numerico e non soddisfa l'espressione `(/d+)`, Django restituisce il solito errore 404 quando l'URL non ha corrispondenze in nessun URLconf.

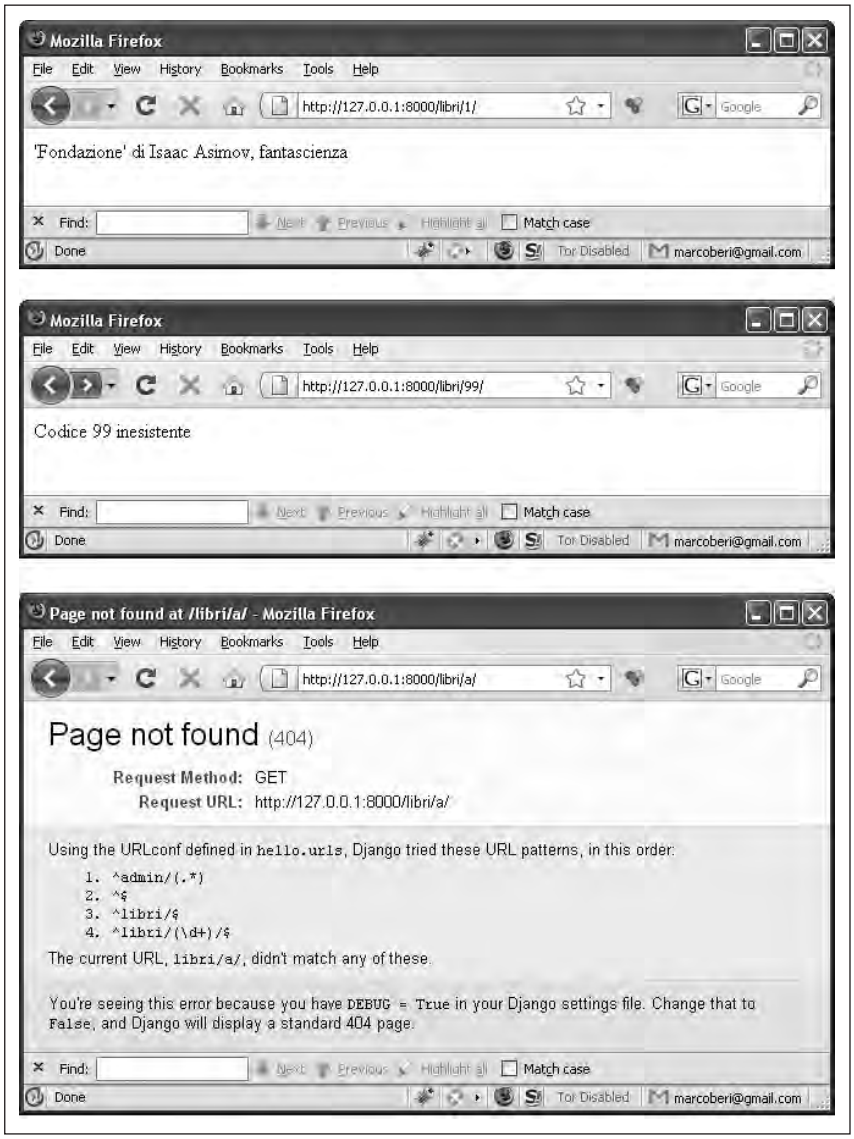

Figura 4.3 Tre link simili con tre risultati diversi.

Parametri con nome

Può succedere che in un URL vogliamo inserire più parametri. Per esempio, ammettiamo di aver inserito nel modello dei libri anche la data di acquisto, e di voler dare la possibilità ai nostri utenti di elencare gli elementi per data. Dobbiamo allora fare in modo che nell'URL siano passati alla vista l'anno e, perché no, anche il mese.

A questo punto, già che ci siamo, inseriamo la data di acquisto nel nostro modello, visto che può capitare di dover modificare uno schema di database già creato. In fondo Django ci aiuta a farlo con davvero poca fatica.

Eseguiamo con attenzione i prossimi passi:

1. Salviamo i dati dell'applicazione `libreria` in un file di testo:

```
python manage.py dumpdata libreria > db.json
```

2. Modifichiamo il modello Libro aggiungendo il campo `data_acquisto` nel file `libreria/models.py`:

```
class Libro(models.Model):
    titolo = models.CharField(max_length=200)
    autore = models.ForeignKey(Autore)
    genere = models.ForeignKey(Genere)
    data_acquisto = models.DateField(null=True,
                    verbose_name="data di acquisto")
```

3. Adesso resettiamo il database con l'apposito parametro `reset`; in questo modo Django ricreerà le tabelle con il nuovo modello che abbiamo appena modificato (rispondiamo tranquillamente `yes` alla domanda che ci viene fatta se siamo sicuri di voler azzerare definitivamente i dati presenti nel database `libreria.db`):

```
python manage.py reset libreria
```

4. Ricarichiamo i dati:

```
python manage.py loaddata data.json
```

Ora proviamo ad aprire il pannello di amministrazione dei libri. Entriamo nella modalità di modifica di un elemento e, se abbiamo fatto tutto bene, dovremmo vedere una finestra come quella nella Figura 4.4. Inseriamo a questo punto delle date per ogni libro.

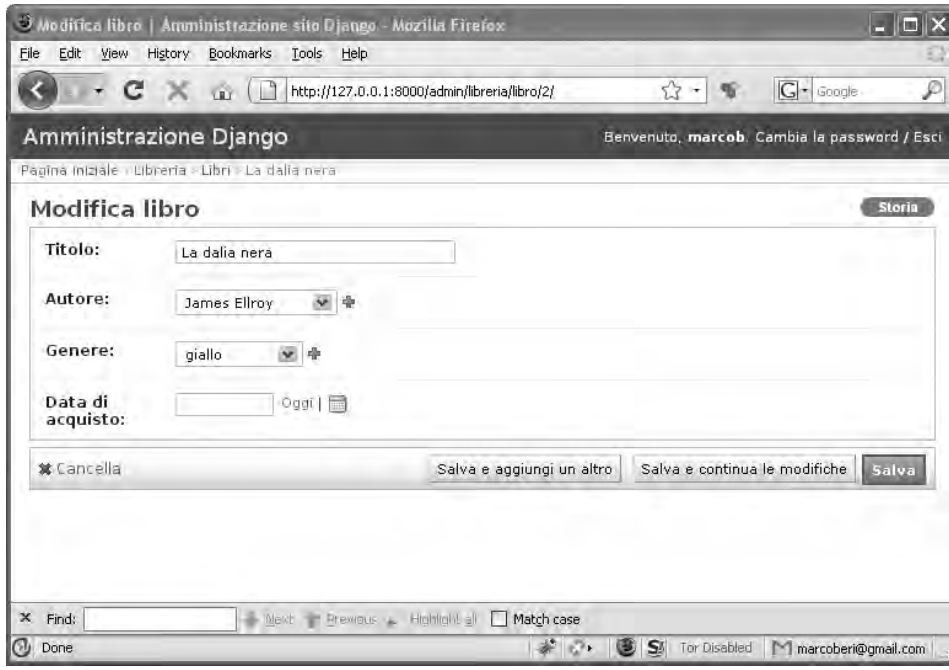

Figura 4.4 Il form di gestione dei libri con il campo Data di acquisto.

Adesso che cominciamo ad avere diversi dati (genere, autore, data di acquisto), la lista dei libri che ci appare nel pannello di amministrazione risulta un po' scarna. Per renderla più amichevole, apportiamo queste modifiche al file `libreria/admin.py`:

```
from django.contrib import admin
from libreria.models import *
class LibroOption(admin.ModelAdmin):
    list_display = ('titolo', 'autore', 'genere', 'data_acquisto')
admin.site.register(Autore)
admin.site.register(Genere)
admin.site.register(Libro, LibroOption)
```

Abbiamo definito una classe `LibroOption` che, attraverso l'attributo `list_display`, indica a Django che dati elencare nella lista dei libri nel pannello di amministrazione.
Questa classe viene passata come secondo parametro dopo la classe `Libro` nella chiamata alla funzione `admin.site.register`. Ora la lista di libri ci apparirà come nella Figura 4.5.

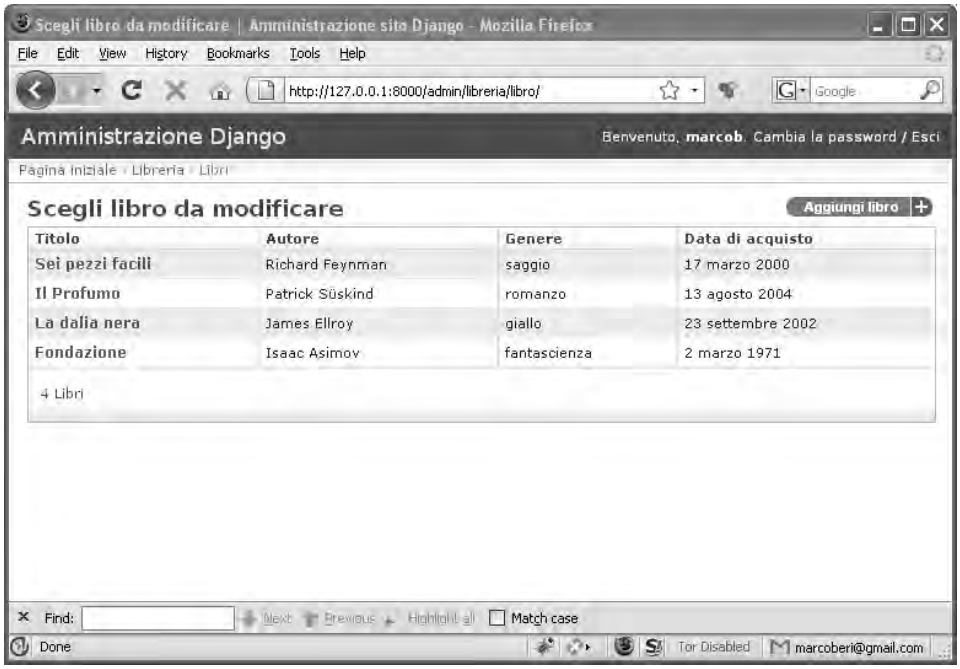

Figura 4.5 Una lista molto più amichevole.

Tornando a monte del discorso iniziato in questo paragrafo, creiamo una vista che ci permetta di selezionare i nostri libri per data di acquisto.
Inseriamo nel file `urls.py` questo elemento:

```
(r'^libri/acquistati/(?P<anno>\d{4})/(?P<mese>\d{1,2})/$',
     "libreria.views.libri_per_data_acquisto")
```

(?P<NOME>...)

La sintassi (`?P<nome>...`) ci permette di dare un nome alla parte di espressione che segue. Nell'URL dell'esempio precedente ci sono quindi due blocchi dal nome `anno` e `mese`.

Django userà i nomi `anno` e `mese` nel passaggio dei valori alla vista. Possiamo ora inserire nel file `libreria/views.py` la seguente funzione, dove abbiamo a bella posta invertito i parametri `mese` e `anno`, rispetto all'ordine dell'URL, per testare se il passaggio dei parametri per nome funziona correttamente:

```
def libri_per_data_acquisto(request, mese, anno):
    libri = Libro.objects.filter(data_acquisto__year=int(anno))
    libri = libri.filter(data_acquisto__month=int(mese))
    elenco = ""
    for libro in libri.order_by('titolo'):
        elenco += "'%s' di %s, %s<br>" % (libro.titolo,
                                    libro.autore, libro.genere)
    if elenco == "":
        elenco = "Nessun libro"
    return HttpResponse(elenco)
```

MA COME? FILTER E POI ANCORA FILTER E INFINE ORDER_BY?

In questa vista abbiamo usato il metodo `filter` del manager `objects` per ottenere un insieme di elementi. Su questo insieme di elementi abbiamo riapplicato un altro `filter` in sequenza. Infine, prima di scorrere la lista, abbiamo specificato con `order_by` l'ordine desiderato. Possiamo in realtà applicare quanti `filter` desideriamo: è del tutto lecito, visto che l'ORM di Django in realtà costruisce la query corretta e completa solo alla fine.

Adesso proviamo ad aprire i seguenti link nel browser per verificare che Django passi i parametri per nome e che la vista li riceva correttamente anche se questi sono in ordine invertito:

- `http://127.0.0.1:8000/libri/acquistati/2004/8/`
- `http://127.0.0.1:8000/libri/acquistati/2002/9/`
- `http://127.0.0.1:8000/libri/acquistati/1971/3/`

Chiaramente, se avete usato delle date di acquisto diverse da quelle nella Figura 4.5, dovrete adattare i precedenti link.

Riepilogo

In questo capitolo abbiamo visto:

- come definire un URLconf nel file `urls.py`;
- come scrivere una semplice vista in corrispondenza dell'URL;
- cos'è la pagina di debug di Django;
- come passare dei parametri dall'URL alla vista;
- come modificare un modello senza perdere i dati;
- come personalizzare la lista dei campi visibili nel pannello di Admin;
- infine, come dare un nome ai parametri passati nell'URL.

Capitolo 5

I template

In questo capitolo

- **URLconf**
- **URL con parametri**
- **Parametri con nome**

"Solo una piccola, sottilissima mentina, signore…"
Il senso della vita – Monty Python

Nel Capitolo 4 abbiamo commesso un vero sacrilegio nell'ottica del diligente *djangonauta*, visto che abbiamo scritto del codice HTML all'interno di una vista e quindi di una funzione Python:

```
for libro in Libro.objects.all(
                        ).order_by('titolo'):
    elenco += "'%s' di %s, %s<br>" % (libro.titolo,
                    libro.autore, libro.genere)
```

Ma non solo: abbiamo anche deciso a livello di vista l'aspetto, il modo di presentare i dati (in inglese si usa la parola *presentation*, che rende abbastanza bene l'idea). La vista, invece, si deve occupare solo di scegliere o comunque trattare i dati da visualizzare.
E allora a chi è delegata la presentation? Ai template (ricordate? Si tratta della "T" di MTV).

Dove scriviamo i template?

Prima di scrivere il primo template, che useremo poi nella vista `libri`, prepariamo un po' il terreno: dobbiamo creare la directory dove conservarlo.
Creiamo una directory `templates` nella directory della nostra applicazione, che avrà ora questa struttura:

```
\hello
     \libreria
              \templates
```

Così facendo siamo a posto, perché Django cerca automaticamente i template in tutte le directory delle applicazioni, alla ricerca di una sottodirectory `templates`.
Se avessimo voluto crearli in un'altra directory, magari perché si trattava di template non specifici di un'applicazione ma di tutto il progetto, avremmo dovuto modificare la variabile `TEMPLATE_DIRS` nel file `settings.py`, aggiungendo il percorso assoluto della directory in questione.
La posizione del nostro progetto potrebbe però cambiare, a seconda se siamo in ambiente di test o in produzione; come possiamo fare per non avere più versioni del file di configurazione? Il file `settings.py` è in tutto e per tutto un modulo Python, per cui possiamo inserire del normalissimo codice. Per esempio, le seguenti righe di Python, messe in cima al file, creano una variabile `ABSOLUTE_PATH` che punta alla directory genitore del nostro progetto:

```
import os
ABSOLUTE_PATH = '%s/' % os.path.abspath(
   os.path.dirname(locals()['__file__'])).replace('\\', '/')
```

Potremmo poi usare questa variabile per modificare il valore di `TEMPLATE_DIRS` facendo in modo che, anche spostando il nostro progetto, tutto funzioni comunque senza problemi:

```
    # TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates"
    # or "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    ABSOLUTE_PATH + 'altra_directory',
)
```

> **BARRA DIRITTA O BARRA RETROVERSA?**
>
> Il commento "Always use forward slashes, even on Windows", ci avvisa di usare sempre la barra diritta (/) anche se stiamo usando Windows, dove normalmente le directory sono separate dalla barra retroversa (\). Proprio per questo, la nostra istruzione che valorizza `ABSOLUTE_PATH` termina con il metodo `.replace('\\', '/')`, che sostituisce le barre retroverse con le barre diritte.

Possiamo ora finalmente scrivere il nostro primo template e usarlo nella vista `libri`.

Il primo template

I template Django sono dei file contenenti codice HTML e istruzioni in un linguaggio definito appositamente dagli autori del framework.
Ci potrà sembrare che il linguaggio sia particolarmente povero (per esempio non è possibile assegnare o cambiare valore di una variabile), ma in realtà questa è stata una precisa scelta progettuale degli autori di Django: i template sono destinati a esprimere l'aspetto, la *presentation*, non la logica di programmazione.
Se ci capiterà di pensare che ci manca un'istruzione per modificare dei dati in un template, ripensiamo con attenzione al design della nostra applicazione: quasi certamente ci accorgeremo di poter fare a meno di quell'istruzione, riprogettando la vista che usa il template.

COSA C'ENTRA IL DESIGN?

Troppo spesso il termine *design* viene associato all'aspetto estetico di un progetto. In realtà non c'è niente di più sbagliato. Leggiamo assieme la definizione di questo termine tratta da un famoso dizionario di italiano: "Design: progettazione di manufatti, da prodursi industrialmente, che contempera le esigenze tecnico-funzionali con quelle estetiche". Quindi il design è progettazione attenta alla funzione di un oggetto e, in un secondo momento, alla sua estetica. I grandi designer sono quelli che progettano una maniglia di una porta che funziona bene e che è anche bella da vedersi. Il design è un concetto che si applica alla perfezione anche alla realizzazione di applicazioni web: che siano belle da vedere ma, soprattutto, che funzionino bene!

Creiamo il file `libreria/templates/libri.html` e inseriamo il seguente codice:

```
<h1>La mia libreria</h1>
<dl>
{% for libro in libri %}
    <dt><strong>"{{ libro.titolo }}"</strong></dt>
    <dd><em>{{ libro.autore }}</em> - {{ libro.genere }}</dd>
    <dd><small>acquistato il {{ libro.data_acquisto }}</small></dd>
{% endfor %}
</dl>
```

Si può notare, scorrendo il template, che abbiamo inserito un ciclo racchiuso tra le istruzioni `{% for ... %}` e `{% endfor %}` e abbiamo visualizzato il contenuto di alcune variabili racchiuse tra `{{` e `}}`. Tutto il resto è semplice codice HTML.
Modifichiamo il file `libreria/views.py` e apportiamo i cambiamenti segnati in grassetto, importando la funzione `render_to_response` e riscrivendo la funzione `libri`:

```
# Create your views here.
from django.http import HttpResponse
from models import *
from django.shortcuts import render_to_response

def libri(request):
    return render_to_response('libri.html', {
        'libri': Libro.objects.all().order_by('titolo')
        })
...
```

La funzione `render_to_response` è una *shortcut* (scorciatoia) che in un colpo solo carica il template, lo *renderizza* e lo restituisce come oggetto `HttpResponse`.
Osserviamo come la funzione `libri` sia ora molto più semplice di prima. Avviamo il server e apriamo con il browser il link `http://127.0.0.1:8000/libri/`. Nella Figura 5.1 il nostro template ha prodotto una lista molto più leggibile, piacevole e completa (oltre che semplice da mantenere) che non la versione precedente.
Ma, un momento, la data in che formato appare? Anno-mese-giorno? Non va bene. Apriamo il template e correggiamo la riga della data aggiungendo la stringa in grassetto:

```
<h1>La mia libreria</h1>
<dl>
{% for libro in libri %}
    <dt><strong>"{{ libro.titolo }}"</strong></dt>
    <dd><em>{{ libro.autore }}</em> - {{ libro.genere }}</dd>
```

Figura 5.1 Il nostro primo template all'opera.

```
<dd><small>acquistato il
   {{ libro.data_acquisto|date:"j F \d\e\l Y" }}</small></dd>
{% endfor %}
</dl>
```

Ci è bastato aggiungere un filtro dove era visualizzata la data di acquisto per modificare il formato di visualizzazione della data. Il filtro `date` è preceduto dalla barra verticale | ed è in questo caso seguito da un parametro `"j F \d\e\l Y"`. Il formato di questo parametro è composto da queste parti (Figura 5.2):

- `j`: giorno in formato numerico (senza lo zero all'inizio);
- `F`: mese in formato testuale e completo;
- `\d\e\l`: stringa `"del"`;
- `Y`: anno in formato numerico e di quattro cifre.

> **TUTTO QUI?**
>
> Per non deludere le aspettative dei più curiosi, consideriamo che ci sono 35 possibili parametri solo per il filtro `date`. Inoltre, oltre a `date`, in Django ci sono altri 54 filtri predefiniti. Anche i più esigenti stiano tranquilli: esistono altre librerie di filtri e, come vedremo, è comunque assai facile scriverne di propri.

Figura 5.2 La seconda versione del template con la data corretta.

Un template più complesso

Stiamo realizzando un'applicazione web e per questo dobbiamo inserire per forza anche qualche link. Per esempio, potrebbe essere interessante fare in modo che, nella lista di libri, si potesse fare clic sul nome dell'autore per essere portati a tutti i libri scritti da questo, e la stessa cosa per quanto riguarda il genere.
Cominciamo a modificare il file `libreria/templates/libri.html`:

```
<h1>La mia libreria</h1>
<dl>
{% for libro in libri %}
    <dt><strong>"{{ libro.titolo }}"</strong></dt>
    <dd><em><a href="/libri/autori/{{ libro.autore.id }}">
    {{ libro.autore }}</a></em> -
    <a href="/libri/generi/{{ libro.genere.id}}">
    {{ libro.genere }}</a></dd>
    <dd><small>acquistato il
    {{ libro.data_acquisto|date:"j F \d\e\l Y" }}</small></dd>
{% endfor %}
</dl>
```

Ora la lista di libri nella Figura 5.3 è certamente più promettente.
Il prossimo passo è quello di modificare il file di URLconf `urls.py`, per aggiungere questi due nuovi URL:

```
 (r'^libri/autori/(\d+)/$', "libreria.views.libri_autore"),
 (r'^libri/generi/(\d+)/$', "libreria.views.libri_genere"),
```

Figura 5.3 Finalmente una pagina web con tanti link!

Ora importiamo questa nuova shortcut e creiamo queste due funzioni nel file `libreria/views.py`:

```
from django.shortcuts import get_object_or_404

def libri_genere(request, id):
    genere = get_object_or_404(Genere, pk=id)
    return render_to_response('libri.html', {
        'libri': Libro.objects.filter(
                genere=genere).order_by('titolo'),
        'genere': genere,})
def libri_autore(request, id):
    autore = get_object_or_404(Autore, pk=id)
    return render_to_response('libri.html', {
        'libri': Libro.objects.filter(
                autore=autore).order_by('titolo'),
        'autore': autore,})
```

Già adesso i nostri link funzionano. Apportiamo ancora qualche ultima miglioria al template `libreria/templates/libri.html`:

```
<h1><a href="/libri/">La mia libreria</a></h1>
{% if genere %}
  <h2>Genere: {{ genere }}</h2>
{% endif %}
{% if autore %}
  <h2>Autore: {{ autore }}</h2>
```

```
{% endif %}
<dl>
{% for libro in libri %}
    <dt><strong>"{{ libro.titolo }}"</strong></dt>
    <dd><em><a href="/libri/autori/{{ libro.autore.id }}">
    {{ libro.autore }}</a></em> -
    <a href="/libri/generi/{{ libro.genere.id}}">
    {{ libro.genere }}</a></dd>
    <dd><small>acquistato il
    {{ libro.data_acquisto|date:"j F \d\e\l Y" }}</small></dd>
{% endfor %}
</dl>
```

Per avere almeno un genere e un autore con più di un elemento, inseriamo il libro della Figura 5.4.

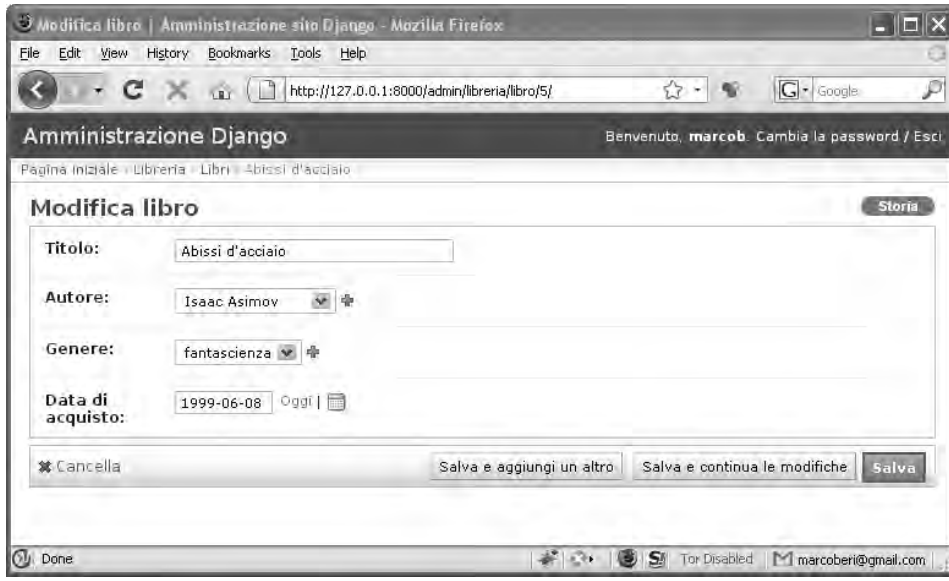

Figura 5.4 Un altro ottimo libro.

Ora possiamo navigare liberamente nella nostra libreria da un genere all'altro, da un autore all'altro e così via (Figura 5.5).

Come estendere un template base

Se sottoponessimo il codice HTML prodotto dalla nostra applicazione a un validatore web (quello ufficiale è `http://validator.w3.org/check`), verremmo insultati dal programma di controllo: manca il `DOCTYPE`, mancano il tag `<HTML>`, il tag `<BODY>` e via dicendo.
Ripetere queste informazioni in tutti i template violerebbe però uno dei principi cardine di Django: DRY, ovvero *Don't repeat yourself !* ("Non ripeterti!").

Figura 5.5 Il genere fantascienza della nostra libreria.

Creiamo quindi un template `base.html` nella directory `libreria/templates`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <title>{% block title %}Hello world!{% endblock %}</title>
</head>
<body>
    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

Questo template costituisce lo scheletro della nostra web application. L'istruzione `{% block <nome blocco> %}` ci permette di definire un'area che possiamo poi sostituire in un template finale.
Modifichiamo adesso il file `libreria/templates/libri.html` in modo da usare ed estendere `base.html`:

```
{% extends "base.html" %}
{% block title %}La mia libreria{% endblock %}
{% block content %}
<h1><a href="/libri/">La mia libreria</a></h1>
{% if genere %}
<h2>Genere: {{ genere }}</h2>
{% endif %}
{% if autore %}
<h2>Autore: {{ autore }}</h2>
{% endif %}
<dl>
```

```
{% for libro in libri %}
    <dt><strong>"{{ libro.titolo }}"</strong></dt>
    <dd><em><a href="/libri/autori/{{ libro.autore.id }}">
    {{ libro.autore }}</a></em> -
    <a href="/libri/generi/{{ libro.genere.id}}">
    {{ libro.genere }}</a></dd>
    <dd><small>acquistato il
    {{ libro.data_acquisto|date:"j F \d\e\l Y" }}</small></dd>
{% endfor %}
</dl>
{% endblock %}
```

L'istruzione `{% extends <nome template> %}` è quella che indica al sistema di template di Django di caricare un template base ed estenderlo con quello corrente.
Per sostituire il contenuto di un blocco del template base dobbiamo usare ancora l'istruzione `{% block <nome blocco> %}`.
Se riapriamo il link `http://127.0.0.1:8000/libri/`, non vediamo alcuna differenza apparente, ma, se copiamo il sorgente della pagina nel validatore W3, il risultato nella Figura 5.6 è più che soddisfacente!

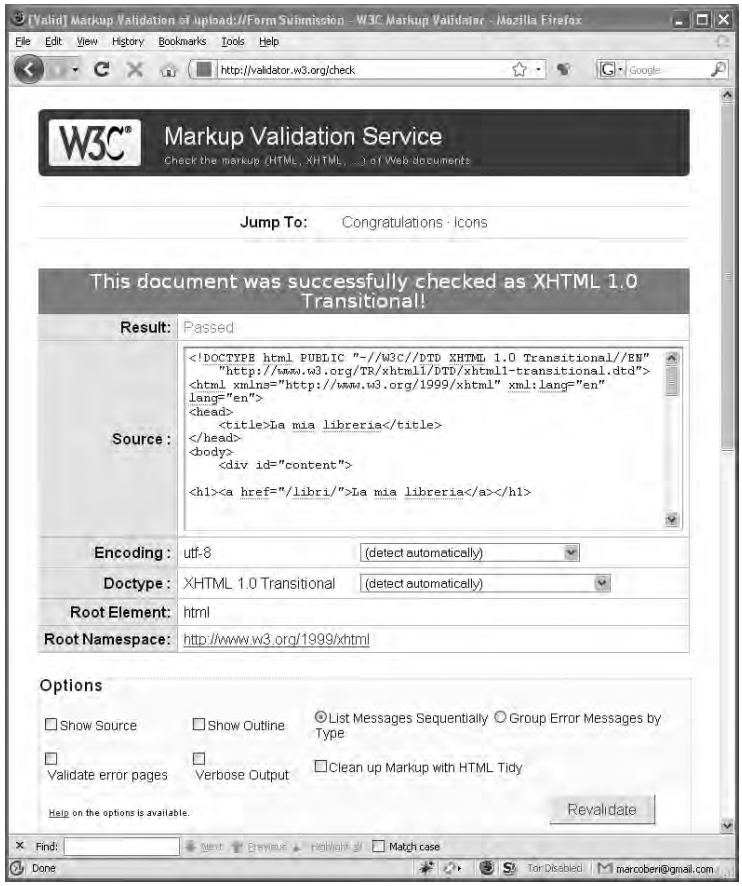

Figura 5.6 La nostra pagina non fa una grinza.

Riepilogo

In questo capitolo abbiamo visto:

- come configurare la posizione dei template;
- come scrivere un template semplice;
- come usare un filtro in un template;
- come scrivere un template più complesso;
- infine, come scrivere un template base da cui derivare, estendendolo, altri template.

Capitolo 6

I form

In questo capitolo

- **Il primo form**
- **Un controllo particolare**
- **Un formato diverso**
- **I test**

"Non avete qualcosa senza lo spam?"
Monty Python's Flying Circus

I form sono praticamente ubiqui in Internet. Non esiste un sito, con un minimo di interazione con i suoi visitatori, che non ne abbia uno o più di uno.
Registrarsi, acquistare, commentare, fare una richiesta: tutte queste e altre operazioni vengono svolte attraverso un form da riempire.
Oltre a essere diffusi, i form sono anche uno degli aspetti più complessi da gestire in un'applicazione web. Ecco alcuni esempi.

- Ci sono form così complicati da dover essere separati su più pagine.
- Ci sono dati inviati dall'utente al server che vanno validati.
- Ci possono essere errori da segnalare che impediscono il salvataggio dei dati stessi e, se questo accade i form, non devono essere azzerati, pena le solenni maledizioni del povero utente che deve ridigitare il contenuto dei campi e che, poco ma sicuro, lo farà solo e soltanto se avrà un grande interesse nel compilare il form, altrimenti addio utente.
- Ci possono essere uno o più file da inviare con i form.

Anche in questo settore Django brilla di luce propria e i suoi form sono molto facili da realizzare e da gestire. A pensarci bene li abbiamo già visti all'opera nel pannello di Admin.

Il primo form

Una funzionalità utile per la nostra libreria potrebbe essere quella di cercare informazioni su Wikipedia per gli autori dei libri inseriti, potendo magari specificare il numero massimo di risultati desiderato o la lingua di Wikipedia in cui effettuare la ricerca.
Questo è esattamente un compito adatto a un form. Inseriamo quindi in `urls.py` l'URL con cui accederemo alla nostra funzione di ricerca:

```
(r'^libri/ricerca/$', "libreria.views_wiki.ricerca"),
```

In futuro potremmo anche costruire un'applicazione apposita, non collegata come questa a un particolare modello, ma per ora limitiamoci a inserire la vista in un file diverso da `views.py`, che chiameremo `views_wiki.py`.
Creiamo il template per la nostra ricerca, il file `wikisearch.html` nella directory `templates`:

```
{% extends "base.html" %}
{% block title %}Ricerca Wikipedia{% endblock %}
{% block content %}
<h1>Ricerca Wikipedia</h1>
<form action="." method="POST">
    {{ form.as_p }}
    <input type="submit" value="Cerca" />
</form>
{% if risultati %}
  <h2>{{ risultati|length }} risultati</h2>
  <ul>
  {% for ris in risultati %}
  <li><a href='{{ link }}{{ ris.title }}'>{{ ris.title }}</a></li>
  {% endfor %}
  </ul>
{% endif %}
{% endblock %}
```

Il template è abbastanza semplice. Estende il template `base.html`, definisce un form che come *action* ha lo stesso URL (`action="."`) e chiama il metodo `as_p` della variabile `form`. Infine, in presenza di una variabile `risultati` valorizzata, la scorre elencandone tutti gli elementi e costruendone i link.
Il secondo e ultimo file che dobbiamo creare è `views_wiki.py`, che salveremo nella directory `libreria`. Il file è un po' più lungo degli altri che abbiamo visto sinora, per cui lo commenteremo un segmento alla volta.
Come prima cosa importiamo tutti i moduli che ci serviranno:

```
import urllib2
from django import forms
from django.shortcuts import render_to_response
from django.shortcuts import get_object_or_404
from django.utils.simplejson.decoder import JSONDecoder
from models import *
```

Alcuni li abbiamo già incontrati, altri sono nuovi. In particolare il modulo `forms` fornisce oggetti e funzionalità per la gestione dei form, mentre l'oggetto `JSONDecoder` ci permette di decodificare stringhe in formato JSON.

COS'È JSON?

In poche parole JSON è come l'XML, solo molto più facile. JSON, acronimo di *JavaScript Object Notation*, è un formato di interscambio per dati leggibile e facilmente editabile, in grado di *serializzare* (trasformare in modo che possano essere trasmesse tra programmi e/o computer diversi) strutture dati anche complesse. Il vantaggio di non richiedere un parser pesante mette JSON in condizioni di essere usato in situazioni limite, come per esempio funzioni JavaScript eseguite dal browser dell'utente. Anche se JSON contiene la parola "JavaScript" nel suo acronimo, ne è completamente indipendente: su `http://json.org` sono a disposizione un centinaio di librerie, per più di trenta linguaggi, per gestire file in formato JSON.

Nella prossima sezione del file `views_wiki.py`, definiamo l'oggetto form con i relativi campi:

```
class WikisearchForm(forms.Form):
    autore = forms.IntegerField(widget=forms.Select(
            choices = [(autore.pk, autore) for autore in
                                Autore.objects.all()]))
    wikipedia = forms.CharField(widget=forms.Select(
                choices=(("it","Italiana"),
                         ("en","Inglese"))))
    limite = forms.IntegerField(initial=10,
                    widget=forms.RadioSelect(
                    choices=((10,"10"),
                             (50,"50"),
                             (100,"100")))
```

Il form `WikisearchForm` contiene tre campi: `autore` (l'autore su cui vogliamo cercare informazioni), `wikipedia` (la lingua di Wikipedia, inglese o italiano) e `limite` (il numero massimo di risultati che vogliamo ottenere).
I primi due sono un intero e una stringa che, grazie al widget `forms.Select`, diventeranno nel nostro form delle liste. Il contenuto di queste liste è dato dal parametro `choices`, che non è altro che un elenco di tuple a due valori (il primo quello "interno" e il secondo quello visualizzato all'utente).
Il valore di `choices` per il campo autore è composto con questa *list comprehension*:

```
[(autore.pk, autore) for autore in Autore.objects.all()]
```

In questo modo nel browser verrà visualizzata la lista dei nomi degli autori, mentre nel form verranno conservate le loro chiavi primarie.
Il terzo campo, infine, è un pulsante d'opzione con tre valori (widget `forms.RadioSelect`), e il valore preselezionato sarà 10 (`initial=10`).

COS'È LA LIST COMPREHENSION?

La *list comprehension* è un costrutto del linguaggio Python (e non solo) che permette di costruire liste, in maniera semplice e intuitiva, a partire da altre liste. Nel nostro caso, partendo dalla lista `Autore.objects.all()`, abbiamo costruito una lista di coppie (<pk>, <nome>). Per essere sinceri, questa domanda non dovrebbe essere fatta da chi conosce Python. ;-)

Le prossime righe del nostro file sono semplicemente due costanti:

```
wiki_url_api="http://%s.wikipedia.org/w/api.php?action=query&"
             "format=json&srlimit=%s&list=search&srsearch=%s"
wiki_link="http://%s.wikipedia.org/wiki/"
```

Useremo la prima costante, `wiki_url_api`, per interrogare le API di Wikipedia. I tre segnaposto `%s` all'interno della stringa verranno sostituiti con i valori scelti tramite il form.
La seconda costante, `wiki_link`, ci serve per comporre, nella lista dei risultati, il link diretto alla pagina di Wikipedia.
Veniamo ora al cuore della nostra vista, la funzione che riceverà i valori dal form ed effettuerà realmente la ricerca su Wikipedia:

```
def ricerca(request):
    risultati = link = None
    if request.method == 'POST':
        form = WikisearchForm(request.POST)
        if form.is_valid():
            autore = get_object_or_404(Autore,
                            pk=form.cleaned_data['autore'])
            url = wiki_url_api % (form.cleaned_data['wikipedia'],
                            form.cleaned_data['limite'],
                            autore.cognome)
            link = wiki_link % form.cleaned_data['wikipedia']
            dati = urllib2.urlopen(url.encode('utf-8')).read()
            valori = JSONDecoder().decode(dati)
            risultati = valori['query']['search']
    else:
        form = WikisearchForm()
    return render_to_response('wikisearch.html', {
        'form': form,
        'link': link,
        'risultati': risultati,})
```

La funzione `ricerca`, in base al valore di `request.method`, è in grado di capire se il form è stato compilato o se abbiamo aperto la vista per la prima volta. In questo ultimo caso presenta il form con i valori di default. Nel caso invece il form sia stato compilato, verifica che i dati siano validi con `form.is_valid()`. Se lo sono, legge dal database l'oggetto `autore` prescelto, compila l'URL da interrogare, legge i dati in formato JSON tornati dalla API di Wikipedia, li decodifica e compila la lista di risultati. Infine, restituisce in risposta il template `wikisearch.html`, passandogli i valori necessari alla sua compilazione.
Arrivati a questo punto, possiamo finalmente aprire con il browser l'indirizzo `http://127.0.0.1:8000/libri/ricerca/` (presumo che oramai sia chiaro che prima è necessario avviare il server di test di Django).
Nella Figura 6.1 vediamo il form pronto per effettuare la ricerca.
Scegliamo l'immortale Isaac Asimov dalla lista degli autori, lasciamo come opzione la Wikipedia in versione italiana e infine optiamo per limitare a 50 elementi i risultati della ricerca.
Facendo clic sul pulsante *Cerca*, dovremmo ottenere quanto mostrato nella Figura 6.2. Notiamo come i campi abbiano mantenuto i valori da noi scelti.
Proviamo ora a selezionare lo scrittore James Ellroy, a limitare a 100 i risultati e infine a effettuare due volte la ricerca sulle due versioni di Wikipedia, prima italiana e poi inglese. La versione italiana dovrebbe restituire 39 risultati, mentre la versione inglese addirittura 100. Abbiamo così verificato il funzionamento di tutti i campi del form.

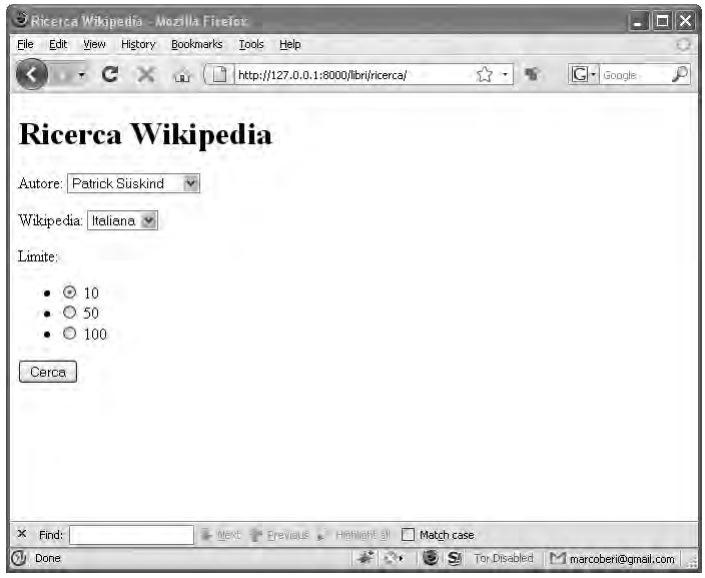

Figura 6.1 Il form per la ricerca su Wikipedia.

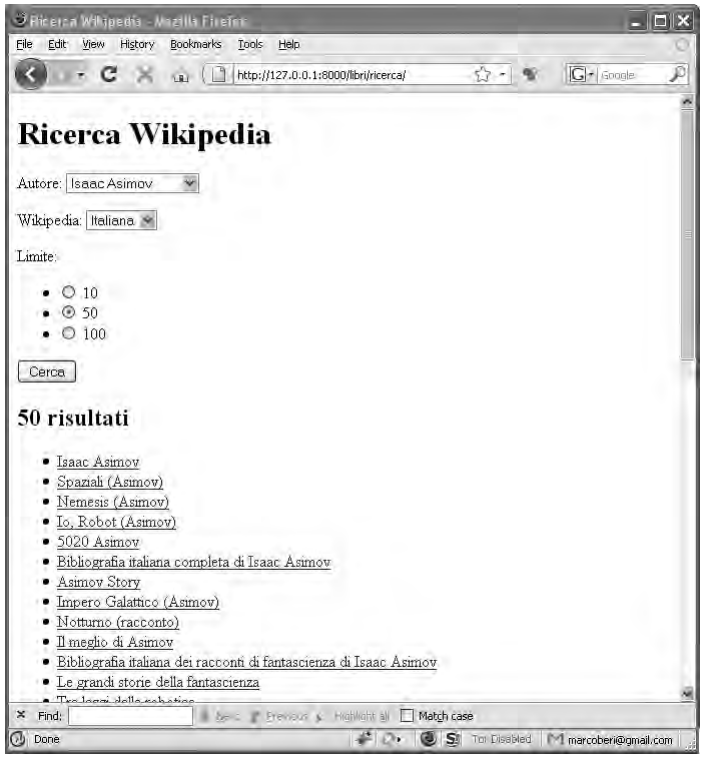

Figura 6.2 Asimov nella versione italiana di Wikipedia.

Un controllo particolare

Il nostro form va benissimo così, ma per provare qualcosa di diverso, immaginiamo, per assurdo, di non voler accettare il valore limite di 100 risultati nel caso della ricerca su Wikipedia in versione inglese.
Apriamo il file `views_wiki.py` e modifichiamo la classe `WikisearchForm` aggiungendo il metodo `clean_limite`:

```
class WikisearchForm(forms.Form):
    autore = forms.IntegerField(widget=forms.Select(
            choices = [(autore.pk, autore) for autore in
                                Autore.objects.all()]))
    wikipedia = forms.CharField(widget=forms.Select(
                choices=(("it","Italiana"),
                         ("en","Inglese"))))
    limite = forms.IntegerField(initial=10,
                    widget=forms.RadioSelect(
                    choices=((10,"10"),
                             (50,"50"),
                             (100,"100"))))
    def clean_limite(self):
        if (self.cleaned_data['limite'] > 50 and
            self.cleaned_data['wikipedia'] == 'en'):
               raise forms.ValidationError(
                       "Massimo 50 risultati in inglese!")
        return self.cleaned_data['limite']
```

Se adesso proviamo a effettuare una ricerca scegliendo Wikipedia in inglese e il limite di 100 risultati, otterremo la risposta visualizzata nella Figura 6.3, con il seguente messaggio subito sotto al campo incriminato: *Massimo 50 risultati in inglese!*.

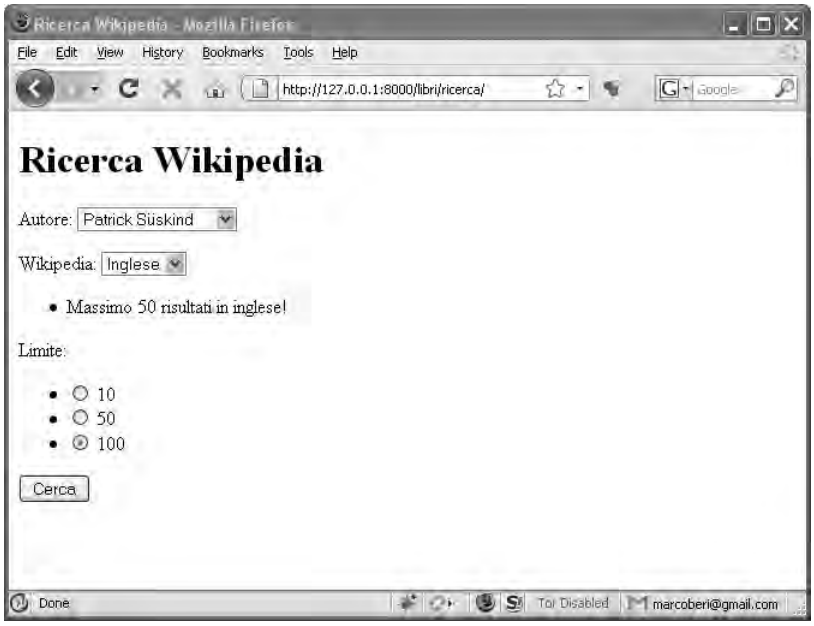

Figura 6.3 Vogliamo troppi risultati da Wikipedia in inglese.

Un formato diverso

Nel template `wikisearch.html` abbiamo usato il metodo `as_p` della variable `form`. In questo modo abbiamo ottenuto tutti i campi come una lista di paragrafi, separati dal tag HTML `<p>`.
Modifichiamo ora la seguente riga:

```
{{ form.as_p }}
```

trasformandola in questo blocco di codice:

```
<table border="1">
   {{ form.as_table }}
</table>
```

Riproviamo ora a causare l'errore del paragrafo precedente (100 risultati con Wikipedia in inglese). La Figura 6.4 mostra il nuovo stile del nostro form.

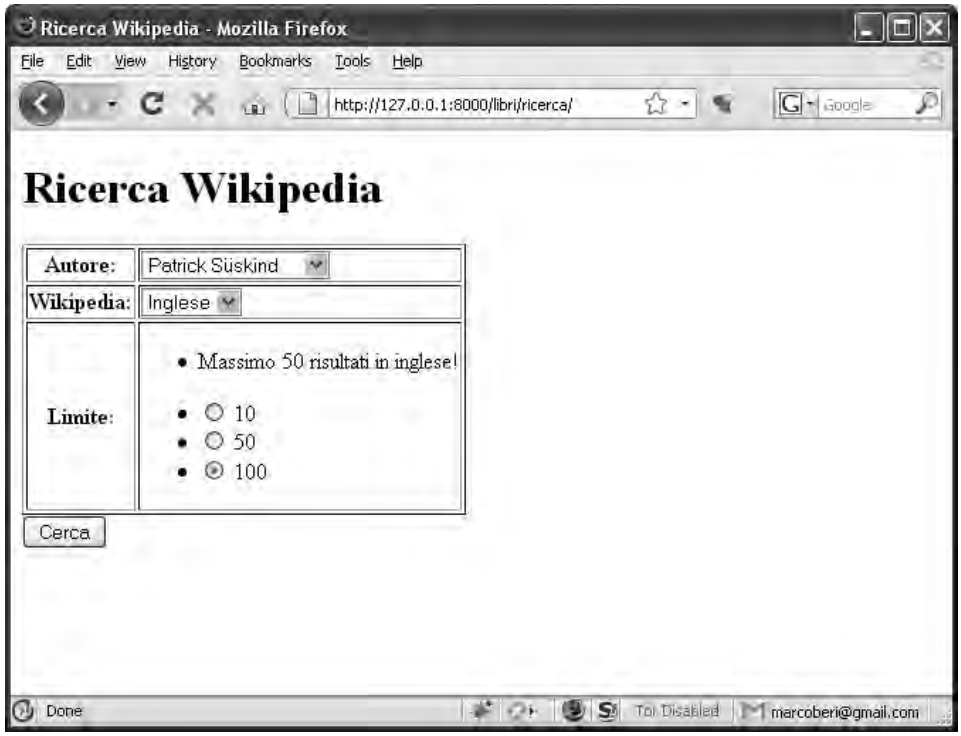

Figura 6.4 Il form in formato tabellare.

Adesso modifichiamo il blocco di codice:

```
<table border="1">
   {{ form.as_table }}
</table>
```

trasformandolo in:

```
<ul>
   {{ form.as_ul }}
</ul>
```

La Figura 6.5 mostra ancora un nuovo stile per il form.

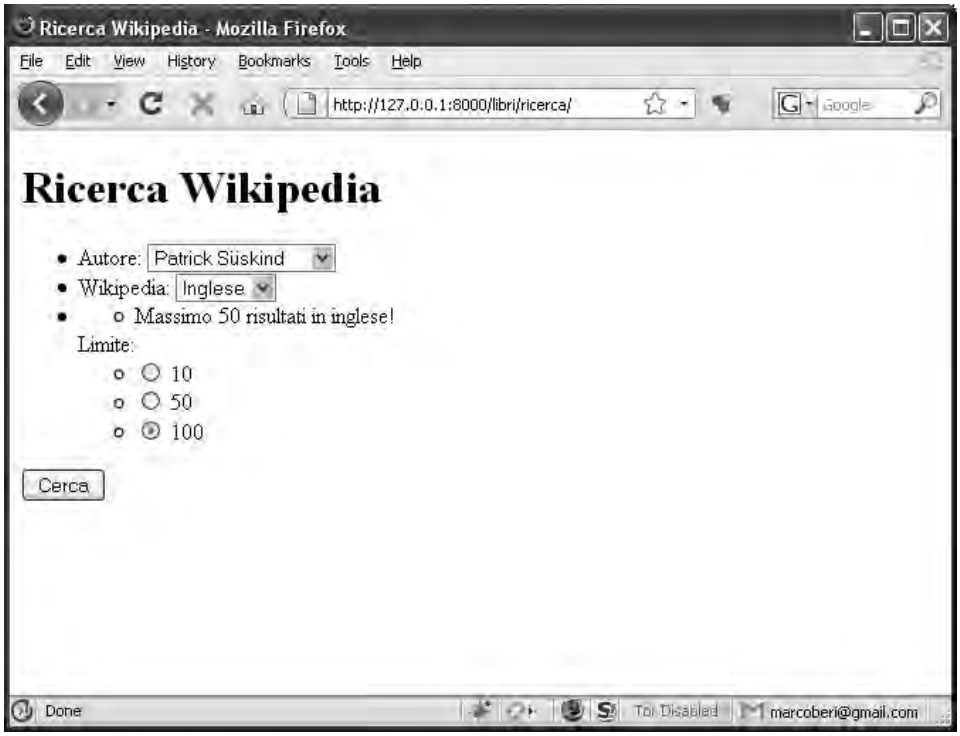

Figura 6.5 Il form in formato lista.

> **E SE NON VOGLIO PARAGRAFI, TABELLE O LISTE?**
>
> Non ci sono solo queste tre possibilità di personalizzazione dello stile del form. In realtà è possibile arrivare a livello di singolo campo. Per esempio, nel nostro template potremmo usare queste tre variabili per accedere ai singoli campi: `form.autore`, `form.wikipedia` e `form.limite`. Per ognuna di esse potremmo accedere ai relativi eventuali errori con `form.autore.errors`, `form.wikipedia.errors` e `form.limite.errors`.

I test

Nei capitoli precedenti abbiamo inserito i test per i modelli della nostra applicazione. Non sarebbe interessante aggiungerli anche per i form? Oltre che interessante è anche fattibile.

Apriamo il file `libreria/tests.py`, creiamo in fondo una funzione cui possiamo dare un nome qualsiasi; nel nostro caso scegliamo `form_test`, e scriviamo il seguente codice:

```
def form_test():
    r"""
>>> from libreria.views_wiki import *
>>> form = WikisearchForm()
>>> form.as_p()
u'<p>...</ul></p>'
>>> form.as_table()
u'<tr><th>...</td></tr>'
>>> form.as_p()
u'<p><label for="id_autore">...</label></li>\n</ul></p>'
>>> form.fields['autore']
<django.forms.fields.IntegerField object at ...>
>>> form.fields['wikipedia']
<django.forms.fields.CharField object at ...>
>>> form.fields['limite']
<django.forms.fields.IntegerField object at ...>
>>> for field in form.fields:
...     print field
...
autore
wikipedia
limite
>>>
"""
```

I tre punti consecutivi (...) posizionati all'interno dell'output atteso si chiamano *ellissi* e indicano alla funzione che esegue il test che in quel punto della stringa può apparire un valore non prevedibile. Per esempio, nella seguente riga, non potendo prevedere in che posizione della memoria sarà creato l'oggetto, usiamo l'ellissi per indicare quello che sarà l'indirizzo reale:

```
<django.forms.fields.IntegerField object at ...>
```

Salviamo il file `tests.py` ed eseguiamo i test:

```
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table libreria_autore
Creating table libreria_genere
Creating table libreria_libro
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for libreria.Libro model
..................
----------------------------------------------------------------------
Ran 18 tests in 1.750s
OK
Destroying test database...
```

In questo modo abbiamo un test in grado di verificare i problemi introdotti da eventuali modifiche al codice di gestione del form per la ricerca.

Riepilogo

In questo capitolo abbiamo visto:

- come definire un form;
- come visualizzarlo in un template;
- come usare i valori compilati nel form in una vista;
- come effettuare una ricerca con le API di Wikipedia;
- come controllare in maniera speciale la correttezza di un dato (con `clean_nomecampo`);
- come modificare lo stile di visualizzazione del form (`form.as_p`, `form.as_table`, `form.as_ul`);
- infine, come aggiungere un test per il form di ricerca.

Capitolo 7

Django e Apache

In questo capitolo

- **Installazione di Apache**

"Se aumentiamo la dimensione di un pinguino fino a che raggiunga la stessa altezza di un uomo e poi confrontiamo le relative dimensioni dei cervelli, scopriremo che il cervello del pinguino è ancora più piccolo. Ma, e questo è il punto, adesso è più grande di 'prima'."
Monty Python's Flying Circus

Usare `python manage.py runserver` in ambiente di produzione è severamente vietato dalla legge. Ovviamente non è così, ma voi fate pure finta che lo sia per davvero.
Django non è un web server, è un web framework e il suo server integrato ha l'unico scopo di facilitare il lavoro degli sviluppatori, che possono, grazie a lui, provare da subito le loro applicazioni.
I web server più conosciuti sono Apache, IIS, lighttpd e nginx. In questo capitolo installeremo il nostro progetto proprio con Apache, visto che al momento è il più diffuso. Infatti a settembre 2008, Netcraft (`http://survey.netcraft.com/Reports/200809/`) indica in circa 91 milioni i siti serviti da Apache contro i 62 milioni serviti da IIS, in seconda posizione e abbastanza distaccato.

Installazione di Apache

Apache, oltre a essere attualmente il server più diffuso sul Web, è anche un progetto open source. Scarichiamo da `http://httpd.apache.org/` la versione adatta al nostro sistema operativo. Per GNU/Linux ci sono i soliti sorgenti in formato `.tar.gz`, mentre per Windows sono disponibili i file di installazione eseguibili.

Apache con GNU/Linux o Mac OSX

A onor del vero con GNU/Linux l'installazione del software è diventata un po' più semplice; per esempio, nelle Figure 7.1 e 7.2 vediamo due programmi grafici per Kubuntu. Se avete scelto l'opzione di scaricare i sorgenti `.tar.gz`, sicuramente sapete già cosa fare (molto in breve, decomprimere, configurare, compilare e installare con i soliti comandi `gzip`, `tar`, `configure` e `make`). La documentazione completa per l'installazione è comunque consultabile all'indirizzo `http://httpd.apache.org/docs/2.2/`.

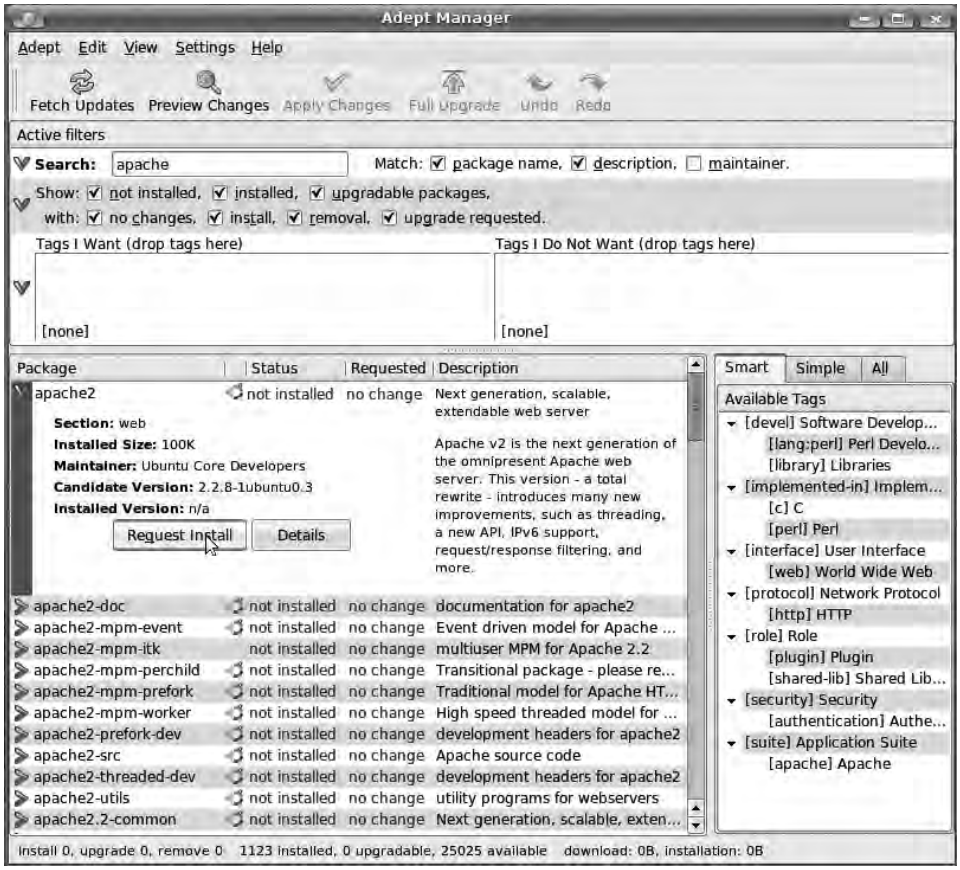

Figura 7.1 Adept Manager.

E PERCHÉ NON USARE WAJIG PER INSTALLARE APACHE?

`wajig` è un programma scritto in Python che semplifica l'uso degli strumenti per la gestione da linea di comando dei pacchetti Debian in GNU/Linux: `apt-get`, `dpkg`, `dpkg-deb`, `apt-cache` e non solo. Già sento i commenti di qualche teknico integerrimo: "Ma come? Sei un pythonista e non usi `wajig`?". Non è vero, ogni tanto lo uso anche io, ma si sa: se vogliamo fare un po' di scena, funzionano meglio i programmi con interfaccia grafica che non quelli da linea di comando.

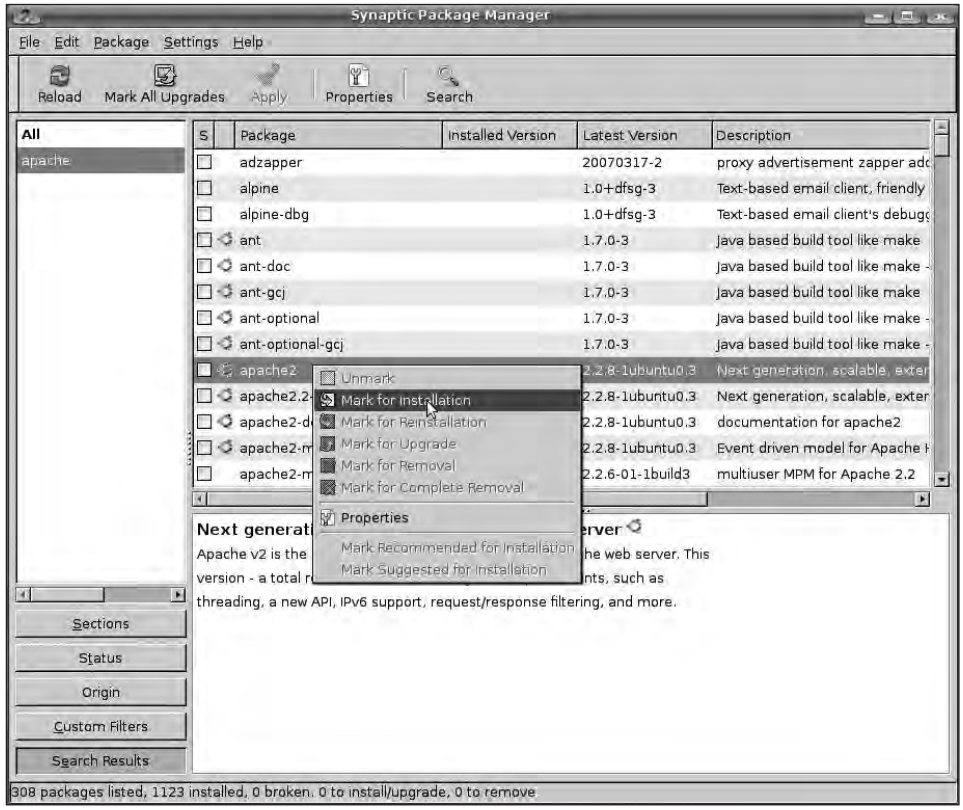

Figura 7.2 Synaptic Package Manager.

Installato Apache, dobbiamo avviarlo e per farlo, normalmente, si usa il seguente comando:

```
sudo /etc/init.d/apache2 start
```

Proviamo ad aprire con il browser l'indirizzo `http://127.0.0.1/` per vedere una semplice pagina con la scritta *It works!* ("Funziona!").
Con Apache funzionante, dobbiamo ora installare `mod_python`, un modulo che permette al nostro web server di eseguire in maniera efficiente programmi Python. Scarichiamo da `http://www.modpython.org/` la versione `.tar.gz` oppure installiamo direttamente con uno degli strumenti di installazione appena visti.
Installato anche `mod_python`, dobbiamo infine configurare Apache per fargli usare Django. Editiamo il file di configurazione, e per farlo il comando dovrebbe essere (il file potrebbe anche chiamarsi `apache.conf` o essere in un'altra directory):

```
sudo vi /etc/apache2/httpd.conf
```

Inseriamo il seguente codice:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    PythonPath "['/home/marcob/', '/home/marcob/hello'] + sys.path"
```

```
    SetEnv DJANGO_SETTINGS_MODULE hello.settings
    PythonDebug On
</Location>
```

Attenzione: dovete modificare la riga in grassetto, quella che definisce `PythonPath`, inserendo sia la directory dove avete creato il progetto `hello`, che abbiamo costruito assieme in questi capitoli, sia la directory del progetto stesso. La prima permette a `mod_python` di trovare `hello.urls` (l'URLconf automaticamente inserito da Django nel file `settings.py`), la seconda consente a Django trovare l'applicazione `libreria` e i suoi moduli.
Potrebbe essere necessario, a seconda della versione di GNU/Linux o Mac OSX, indicare espressamente ad Apache di caricare il modulo `mod_python`, inserendo nel suo file di configurazione anche il seguente comando:

```
LoadModule python_module libexec/mod_python.so
```

Facciamo ricaricare ad Apache il file di configurazione:

```
sudo /etc/init.d/apache2 reload
```

Ora possiamo aprire con il browser l'indirizzo `http://127.0.0.1/libri/` per vedere la lista dei nostri libri servita da Apache con l'aiuto di Django.

Apache con Windows

Se invece il nostro sistema operativo è Windows, avremo scaricato il relativo pacchetto di installazione binario. Eseguendolo, dovremmo vedere la finestra mostrata nella Figura 7.4. Premiamo il pulsante *Next*, accettando i termini della licenza (una delle più permissive in assoluto) e, quando arriviamo alla finestra mostrata nella Figura 7.6, scegliamo un nome appropriato per il dominio e per il server (nella figura sono inseriti dei nomi di fantasia). Quindi, facendo clic altre volte su *Next*, completiamo l'installazione.
Dovremmo a questo punto vedere apparire una icona a forma di piuma nel nostro system tray, come quella all'estrema sinistra della Figura 7.3, qui sotto. La freccia verde segnala che Apache è in esecuzione. Proviamo ad aprire con il browser l'indirizzo `http://127.0.0.1/` per vedere una semplice pagina con la scritta *It works!* ("Funziona!").

Figura 7.3 Il system tray con a sinistra l'icona di Apache.

Come abbiamo già detto nel paragrafo precedente, dobbiamo ora installare `mod_python`, il modulo che permette ad Apache di eseguire in maniera efficiente programmi Python.
Scarichiamo da `http://www.modpython.org/` la versione binaria eseguibile per Windows e avviamola. L'unica domanda che ci viene fatta in fase di installazione è quella che vediamo nella Figura 7.7, in cui `mod_python` vuole sapere dove abbiamo installato Apache sul nostro sistema.
Al termine dell'installazione non ci resta che modificare i file di configurazione di Apache per indicargli di usare Django.
Per modificare il file di configurazione, durante l'installazione di Apache è stata creata un'apposita voce di menu, ovvero *Edit the Apache httpd.conf Configuration File*, come vediamo anche nella Figura 7.5.

Figura 7.4 La prima schermata di installazione di Apache in Windows.

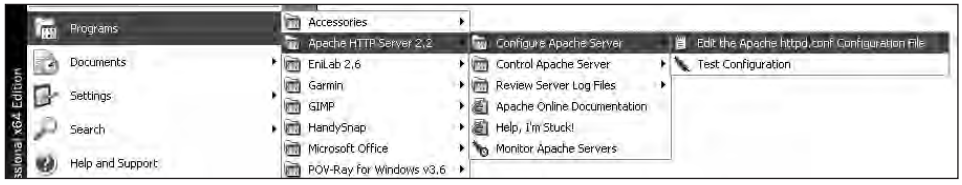

Figura 7.5 Come modificare il file di configurazione di Apache.

Figura 7.6 La schermata di configurazione di Apache.

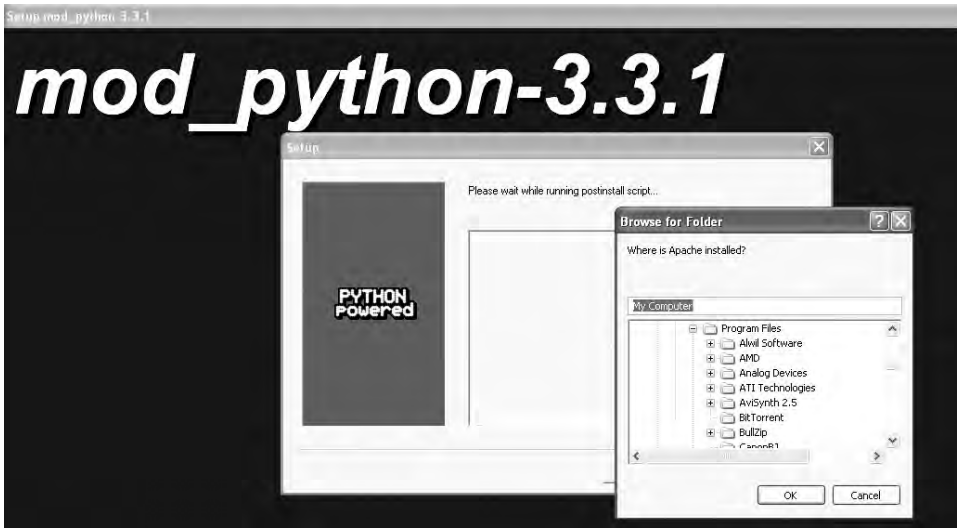

Figura 7.7 mod_python chiede dove è installato Apache.

Il comando di modifica del file di configurazione (*Edit*) aprirà il file `httpd.conf` con il vostro editor di testo predefinito (in genere Blocco note).
Inseriamo le seguenti righe di codice:

```
LoadModule python_module modules/mod_python.so
<Location "/">
    SetHandler python-program
    PythonPath "['C:\work', 'C:\work\hello'] + sys.path"
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE hello.settings
    PythonDebug On
</Location>
```

Dovete modificare le directory della riga in grassetto, adattandole alla posizione dove avete creato il progetto `hello`.
L'ultima modifica riguarda il file `settings.py`. Nel Capitolo 5 abbiamo parlato di una variabile `ABSOLUTE_PATH` che poteva servirci.
Ebbene, è arrivato il momento di usarla e di modificare quindi `settings.py`, aggiungendo queste righe:

```
import os
ABSOLUTE_PATH = '%s/' % os.path.abspath(
   os.path.dirname(locals()['__file__'])).replace('\\', '/')
```

Usiamo ora la variable `ABSOLUTE_PATH` per definire in maniera assoluta la posizione del nostro database:

```
DATABASE_NAME = ABSOLUTE_PATH + "libreria.db"
```

Perché abbiamo dovuto fare questa modifica del nome del database? Il server integrato di Django viene eseguito direttamente nella directory del nostro progetto, per cui possiamo permetterci di lasciare il solo nome del file corrispondente al database SQLite. Quando invece le nostre pagine sono servite da Django attraverso Apache, questo non è più possibile,

perché sia Apache sia Django si trovano in altre directory. Se vogliamo evitare un errore di database `libreria.db` non trovato, dobbiamo inserire il pathame assoluto. Dopo queste modifiche dobbiamo riavviare Apache per fargli ricaricare i nuovi file di configurazione. Facendo clic con il pulsante destro del mouse sull'icona nel system tray, scegliamo *Open Apache Monitor*. La Figura 7.8 mostra la finestra del monitor: premiamo sul pulsante *Restart*. Aprendo con il browser l'indirizzo `http://127.0.0.1/libri/` possiamo vedere la lista dei nostri libri.
Se usiamo un add-on per Firefox che ci permetta di vedere gli header, possiamo verificare che la pagina sia servita da Apache (come per esempio nella Figura 7.9).

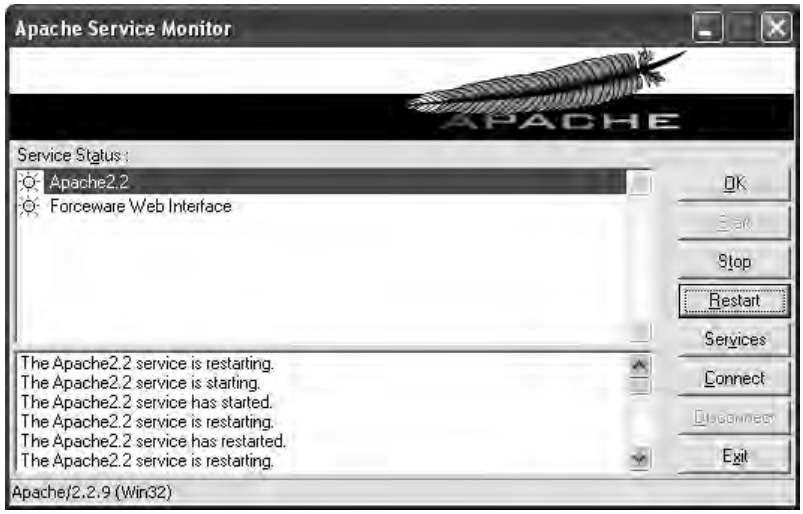

Figura 7.8 Il monitor di Apache.

Figura 7.9 Gli header visualizzati con l'add-on per Firefox Web Developer.

Riepilogo

In questo capitolo abbiamo visto:

- come installare Apache su GNU/Linux o Window;
- come installare e configurare `mod_python`;
- infine, come configurare Apache per servire pagine dinamiche con Django.

Capitolo 18

Navighiamo tra i dati (databrowse)

In questo capitolo

- **Attivare databrowse**
- **Proteggere l'accesso di databrowse**

"Mi fa male il cervello!"
Sketch "Gumby Brain Specialist"
– Monty Python's Flying Circus

Se il pannello Admin automatico vi è piaciuto, `databrowse` vi stupirà! Si tratta di un'applicazione già compresa in Django che ci permette di navigare liberamente tra i dati del nostro database.
Questa ricetta è composta da poche righe di codice e molte immagini: guardandole capirete perché.

Attivare databrowse

Per attivare `databrowse` dobbiamo inserire tra le `INSTALLED_APPS` in `settings.py` la seguente riga:

```
'django.contrib.databrowse',
```

Quindi in `urls.py` inseriamo lo schema di URL di `databrowse`:

```
from django.contrib import databrowse
urlpatterns = patterns('',
(r'^databrowse/(.*)', databrowse.site.root),
...
```

Infine dobbiamo registrare i modelli in cui vogliamo navigare. Possiamo farlo inserendo queste altre righe sempre in `urls.py`:

```
from libreria.models import *
databrowse.site.register(Genere)
databrowse.site.register(Autore)
databrowse.site.register(Libro)
databrowse.site.register(Articolo)
```

Ora visitiamo l'indirizzo `http://127.0.0.1:8000/databrowse/` e... sorprendiamoci!
Le figure in fondo a questa ricetta sono un viaggio tra i dati della libreria: seguite il puntatore del mouse.

Proteggere l'accesso di databrowse

Per proteggere l'accesso a `databrowse` sono sufficienti queste semplici modifiche all'URLconf, che provocheranno la richiesta di nome utente e password:

```
from django.contrib.auth.decorators import login_required
(r'^databrowse/(.*)', login_required(databrowse.site.root)),
```

Dobbiamo poi accertarci che tra gli schemi di URL ci sia anche quello della schermata di login:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login'),
```

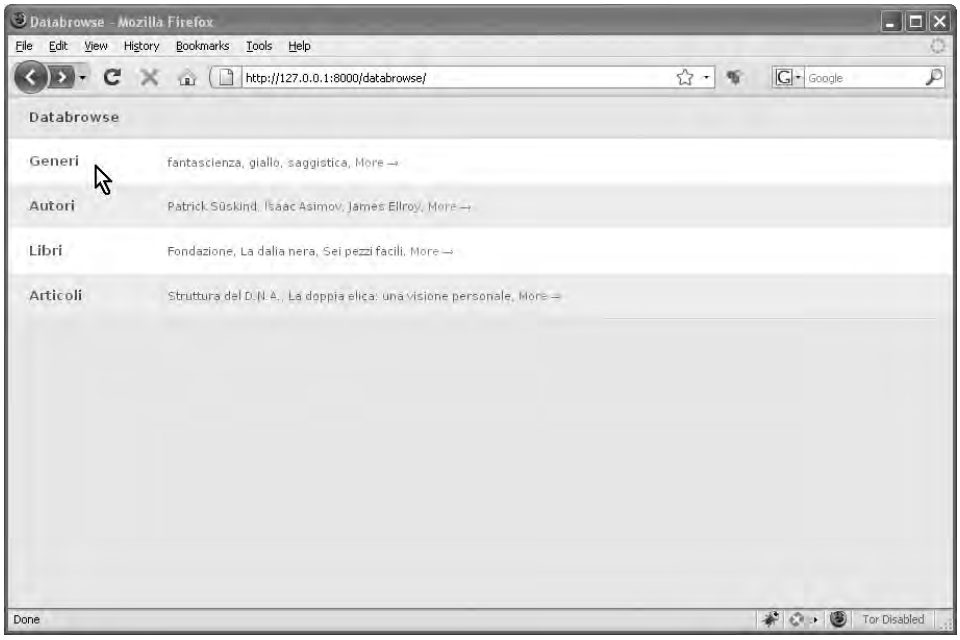

Figura 18.1 La homepage di databrowse, con tutti i modelli che abbiamo registrato.

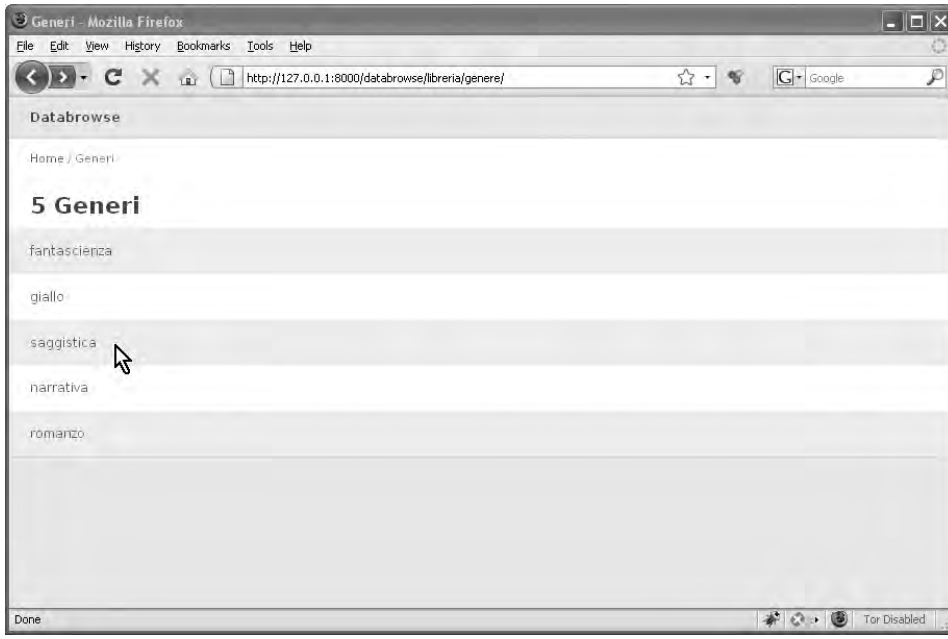

Figura 18.2 I generi della nostra libreria.

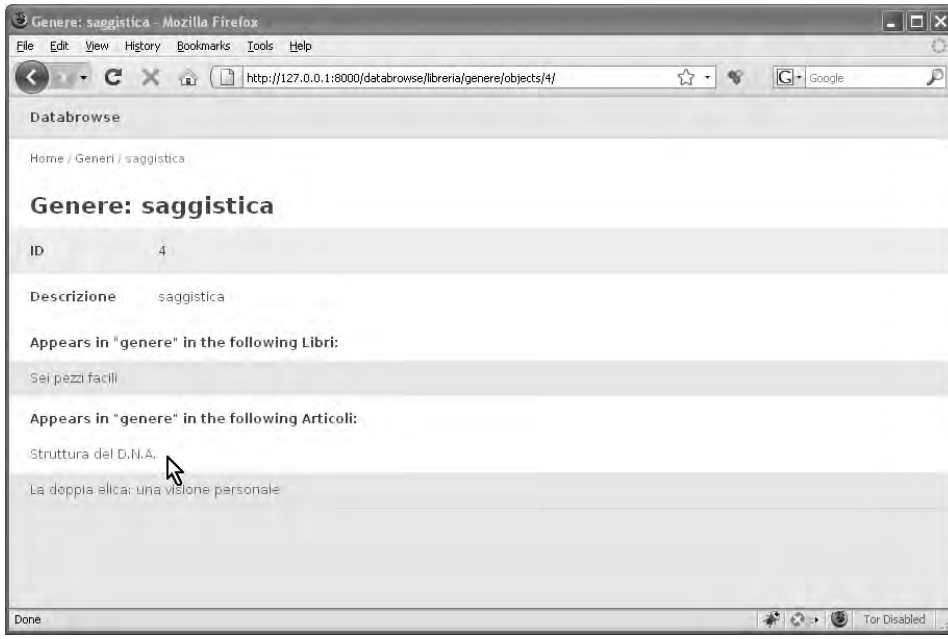

Figura 18.3 Il genere "Saggistica".

Figura 18.4 L'articolo "Struttura del D.N.A.".

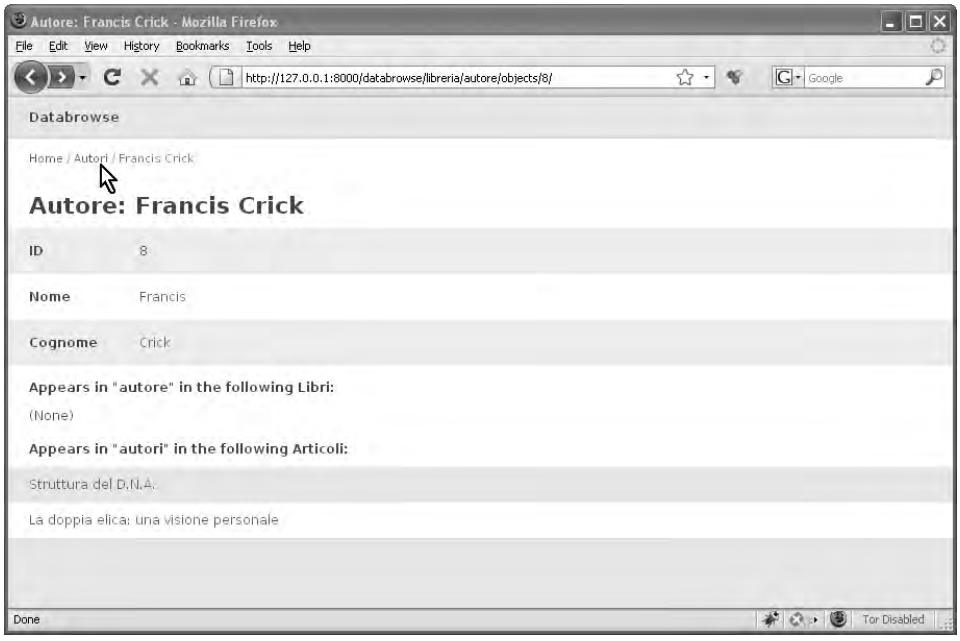

Figura 18.5 Uno dei suoi autori: Francis Crick.

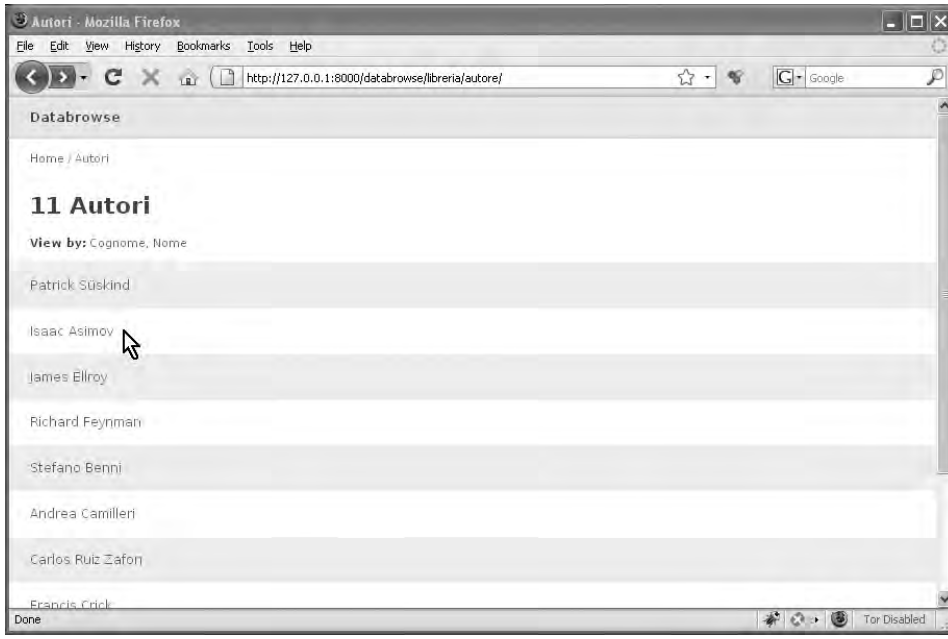

Figura 18.6 Gli autori.

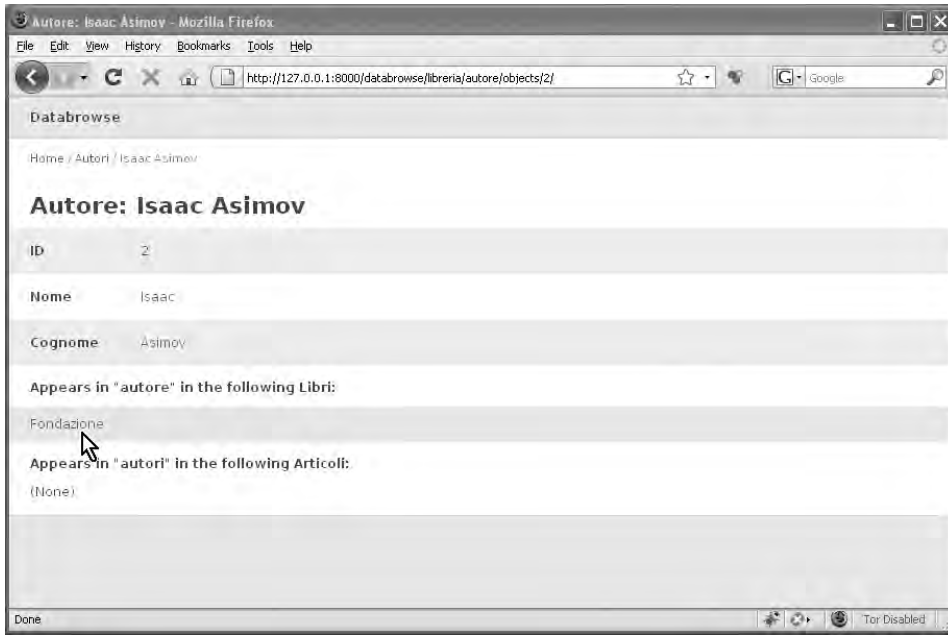

Figura 18.7 Il grandissimo Isaac Asimov.

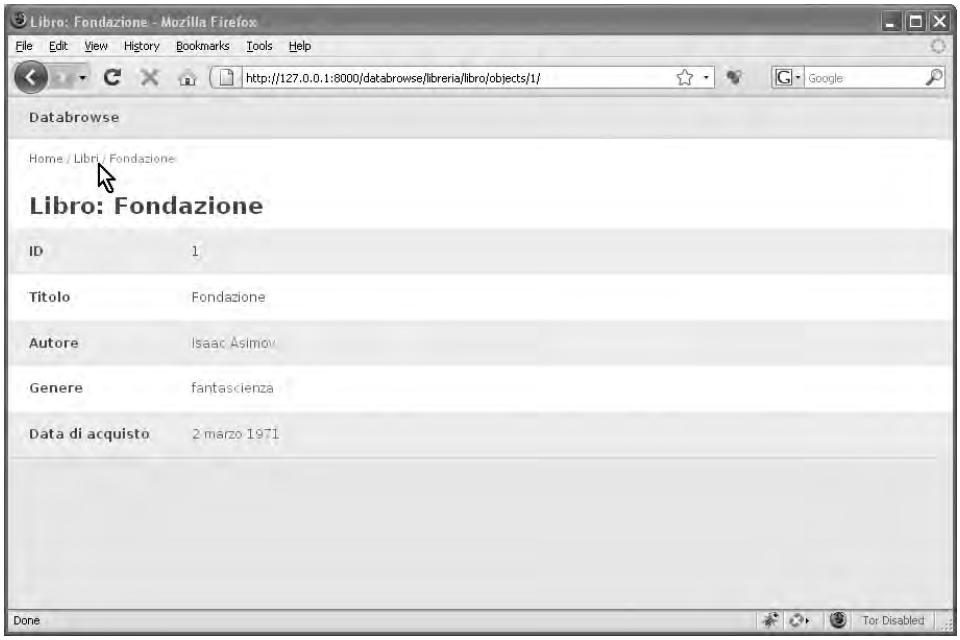

Figura 18.8 *Fondazione*, uno dei suoi più bei libri.

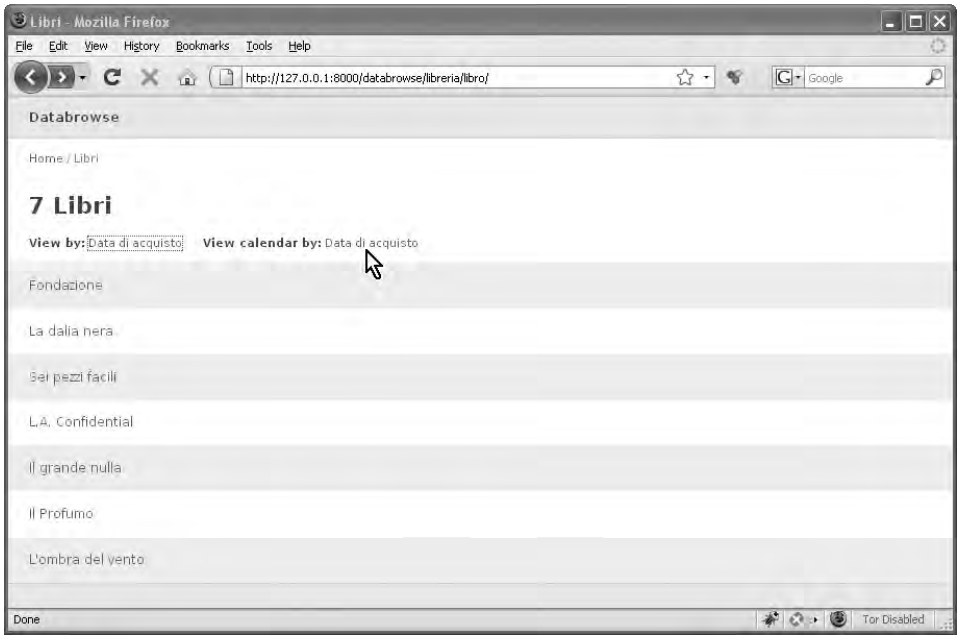

Figura 18.9 I libri e in alto l'accesso ai possibili ordinamenti.

Figura 18.10 I libri per data di acquisto.

Figura 18.11 I mesi del 2000 in cui abbiamo acquistato dei libri.

Riepilogo

In questa ricetta abbiamo visto come:

- attivare `databrowse`;
- infine, proteggere l'accesso di `databrowse`.

Capitolo 19

Uscire dal seminato (generare file non HTML)

"Al momento sto lavorando su una nuova malattia, da cui spero di trarre un musical."
Sketch "Thripshaw's disease" – Monty Python's Flying Circus

In un esempio del Capitolo 13, abbiamo provato a inviare al browser un file Excel. In effetti si trattava semplicemente di dati in formato testo CSV, con un `content_type` apposito.
In questa ricetta proveremo a generare un file in formato PDF, un'impresa solo un po' più difficile della precedente. In particolare convertiremo in formato PDF la pagina HTML della lista dei nostri libri, così come appare nella Figura 19.1.

In questo capitolo

- **Installare le librerie necessarie**
- **Modificare la vista per generare un file PDF**
- **Attivare la generazione del file PDF**

Installare le librerie necessarie

Come per tutti gli altri esempi di questo libro, useremo solo librerie liberamente disponibili su Internet.
Per questa ricetta dobbiamo scaricare tre componenti.

- ReportlabToolkit, per creare file PDF (da `http://www.reportlab.org`).
- html5lib, per interpretare il codice HTML (da `http://code.google.com/p/html5lib/`).
- pisa, per convertire da HTML a PDF (da `http://pypi.python.org/pypi/pisa/`).

È scontato ma lo dico lo stesso: si tratta in tutti e tre i casi di librerie per Python.

Figura 19.1 La lista dei nostri libri.

Modificare la vista per generare un file PDF

Nel Capitolo 5 abbiamo creato la funzione che genera la lista di libri, nel file `hello/libreria/views.py`:

```
def libri(request):
    return render_to_response('libri.html', {
        'libri': Libro.objects.all().order_by('titolo')
        })
```

Modifichiamola nel seguente modo:

```
import ho.pisa, cStringIO
def libri(request, tipo=None):
    libri = Libro.objects.all().order_by('titolo')
    response = render_to_response('libri.html',
                RequestContext(request, {'libri': libri, }))
    if tipo == ".pdf":
        file_pdf = cStringIO.StringIO()
        ho.pisa.CreatePDF(unicode(response.content,
                          encoding='utf-8'), file_pdf)
        return HttpResponse(file_pdf.getvalue(),
                            content_type=home pageapplication/pdf")
    return response
```

Se visitiamo l'indirizzo `http://127.0.0.1:8000/libri/`, non dovremmo notare alcuna differenza rispetto a quanto mostrato nella Figura 19.1.

Attivare la generazione del file PDF

La nuova versione della vista `libri` accetta un parametro `tipo`:

```
def libri(request, tipo=None):
```

Modifichiamo, in `urls.py`, il corrispondente schema URL:

```
(r'^libri(?P<tipo>\.pdf)?/?$', "libreria.views.libri"),
```

Se adesso visitiamo l'indirizzo `http://127.0.0.1:8000/libri.pdf`, vedremo quanto mostrato nella Figura 19.2.

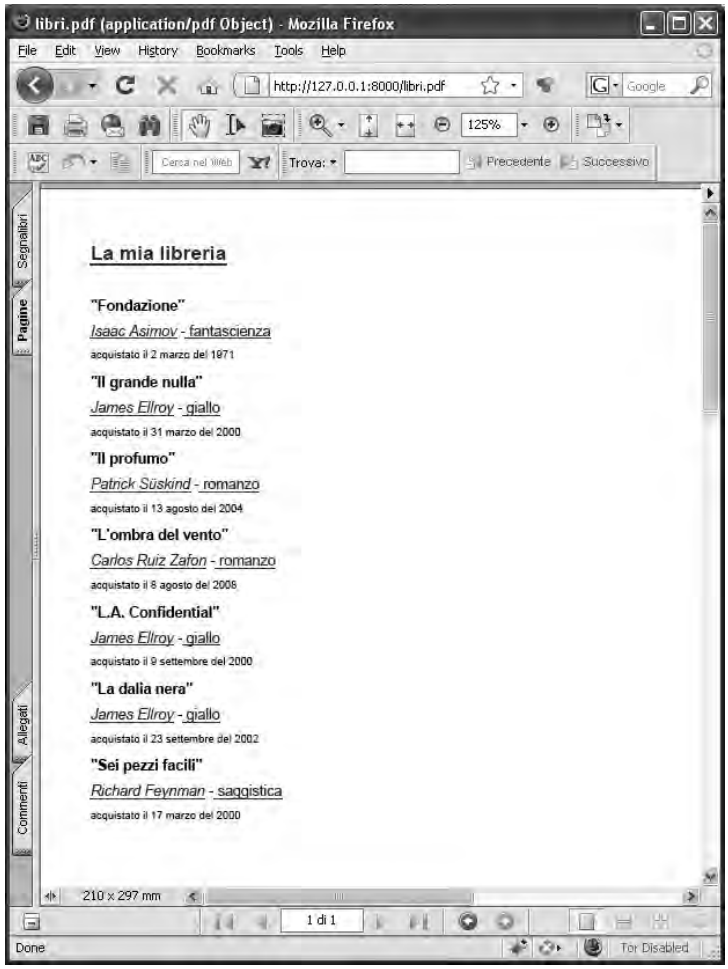

Figura 19.2 Il file PDF con la lista dei nostri libri.

Per attivare più conversioni è sufficiente modificare l'URL aggiungendo altri tipi:

```
(r'^libri(?P<tipo>\.xls|\.pdf|\.csv)?/?$', "libreria.views.libri"),
```

Gli URL riconosciuti saranno a questo punto questi quattro:

- `http://127.0.0.1:8000/libri/`
- `http://127.0.0.1:8000/libri.xls`
- `http://127.0.0.1:8000/libri.pdf`
- `http://127.0.0.1:8000/libri.csv`

> **POSSO CREARE UN FILE EXCEL IN FORMATO NATIVO?**
>
> Ebbene sì. Se avete la sventura di dover generare a tutti i costi un file in formato nativo Excel, esiste una libreria Python che fa per voi (ovviamente sempre open source). La potete trovare all'indirizzo `http://pypi.python.org/pypi/xlrd/`.

Riepilogo

In questa ricetta abbiamo visto come:

- installare le librerie necessarie per la conversione in file PDF;
- modificare la vista per convertire da HTML a PDF;
- infine, attivare la generazione del file PDF.

Capitolo 20

Pulito sì, fatica no (Ajax)

"E alla mia destra, a sostenere le ragioni contrarie al governo, una piccola chiazza di liquido marrone."
Sketch "Face the press"
– Monty Python's Flying Circus

Termini come *social network*, *tagsonomy* ("tag-sonomia", in contrapposizione a tassonomia) e *folksonomy* ("tagging sociale") sono spesso usati a sproposito da commerciali senza scrupoli che vogliono impressionare i loro (futuri) clienti. In realtà, questa ondata di novità, che possiamo catalogare sotto il nome di Web 2.0, qualcosa di buono l'ha portato, soprattutto sul fronte tecnologico: Ajax, per esempio. Grazie ad Ajax possiamo infatti realizzare interfacce molto più accattivanti e "amichevoli", rispetto ai freddi form HTML del secolo scorso.
In questa ricetta, proveremo a estendere l'interfaccia di Admin, in modo da permettere la modifica del titolo di un libro direttamente nella vista (quella con l'elenco degli elementi da modificare).

In questo capitolo

Installare jQuery

La libreria che useremo per questa ricetta è la fantastica jQuery. Personalmente, ritengo che jQuery stia alle varie librerie JavaScript come Django sta ai framework web.
La versione attuale è `jquery-1.2.6.min.js`. Dopo averla scaricata da `http://jquery.com/`, va copiata nella directory `hello/content/js`.

Creare la colonna per la modifica

Modifichiamo `libreria/models.py` aggiungendo al modello `Libro` il seguente metodo:

```
def modifica_titolo(self):
    return (u'<input type="text" name="%s" value="%s" size="30"'
            'class="modifica-titolo">' % (self.pk, self.titolo))
modifica_titolo.allow_tags = True
```

Il metodo restituisce il codice HTML per un campo di input, avente l'attributo `name` uguale alla chiave primaria dell'elemento corrente, con classe `modifica-titolo` e contenuto uguale all'attributo `titolo`.

Visualizzare la colonna

Ora dobbiamo modificare l'attributo `list_display` della classe `LibroOption`, nel file `libreria/admin.py`, aggiungendo il valore `'modifica_titolo'`:

```
list_display = ('titolo', 'autore', 'genere',
                'data_acquisto', 'modifica_titolo')
```

A questo punto, se visualizziamo la lista di libri da modificare nel pannello di Admin all'indirizzo `http://127.0.0.1:8000/admin/libreria/libro/`, dovremmo vedere la stessa schermata della Figura 20.1.

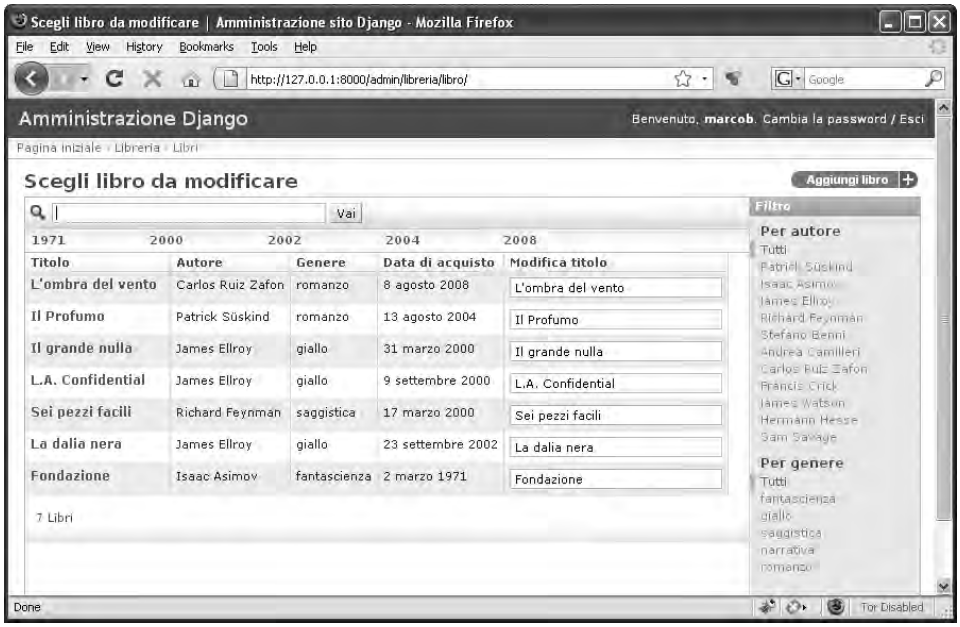

Figura 20.1 La colonna dei titoli modificabili.

Creare la funzione Ajax

Creiamo il file `hello/content/js/modifica-titolo.js` con il seguente contenuto:

```
$(document).ready(function() {
    $('.modifica-titolo').css("margin", "0");
    $('.modifica-titolo').change(function(){
        var input = $(this);
        $.ajax({
            url: "./"+ input.attr('name') +"/titolo/",
            data: "titolo="+input.attr('value'),
            type: "POST",
            complete: function(xmlHttpRequest, message){
                if (message == 'success') {
                    input.css("border", "1px solid green");
                } else {
                    input.css("border", "1px solid red");
                }
            },
        });
    });
});
```

Questa funzione viene chiamata al caricamento della pagina completata, grazie all'evento `ready` del documento (`$(document).ready`), e aggancia una funzione Ajax all'evento `onChange` di tutti i campi con classe `modifica-titolo`.
La funzione Ajax effettua una `POST` a un URL apposito `"./<pk>/titolo/"` del valore del titolo modificato. In caso di successo colora di verde il contorno del campo, mentre in caso di errore lo colora di rosso.

Inserire jQuery e la funzione Ajax nel template

Ora dobbiamo fare in modo che la nostra funzione, insieme alla libreria jQuery, venga caricata dal browser nella schermata di Admin. Per riuscirci, basta creare un template con il nome seguente:

`hello/libreria/templates/admin/libreria/libro/change_list.html`

Il contenuto del template è il seguente:

```
{% extends "admin/change_list.html" %}
{% block extrahead %}
<script type="text/javascript"
   src="{{ MEDIA_URL }}/js/jquery-1.2.6.min.js"></script>
<script type="text/javascript"
   src="{{ MEDIA_URL }}/js/modifica-titolo.js"></script>
{% endblock %}
```

Per avere a disposizione la variabile `MEDIA_URL` nel template, dobbiamo aggiungere, nel file `settings.py`, il context processor `"django.core.context_processors.media"` alla variabile `TEMPLATE_CONTEXT_PROCESSORS`.

Creare la vista di salvataggio

Ora dobbiamo creare la vista che verrà chiamata dalla funzione Ajax. Inseriamo il seguente codice nel file `hello/libreria/views.py`:

```
from django.contrib.auth.decorators import login_required
from django.views.decorators.http import require_POST
from django.http import HttpResponse, HttpResponseNotModified
from django.http import HttpResponseBadRequest
from django.shortcuts import get_object_or_404
@login_required
@require_POST
def modifica_titolo(request, libro_pk):
    if request.is_ajax and request.user.is_staff:
        titolo = unicode(request.POST.get('titolo')).strip()
        if not titolo or Libro.objects.exclude(pk=libro_pk
                  ).filter(titolo__iexact=titolo).count() > 0:
            return HttpResponseBadRequest(content='Ko'.)
        libro = get_object_or_404(Libro, pk=libro_pk)
        if libro.titolo != titolo:
            libro.titolo = titolo
            libro.save()
            return HttpResponse(content='Ok')
        return HttpResponseNotModified(content='Ok')
    return HttpResponseBadRequest(content='Ko')
```

La funzione può essere richiamata solo in `POST` e solo dopo che l'utente ha effettuato il login (questo grazie ai due decoratori `login_required` e `require_post`). Quindi controlliamo che la `request` sia di tipo Ajax e che l'utente abbia accesso all'Admin (`request.is_ajax` e `user.is_staff`). A questo punto effettuiamo qualche controllo sul valore di `titolo` (che non sia vuoto, un doppione o uguale al titolo già presente) e quindi lo salviamo.

Attivare la vista

L'ultimo passaggio è l'attivazione della vista, grazie all'inserimento di uno schema di URL nell'URLconf `hello/urls.py`:

```
(r'^admin/libreria/libro/(?P<libro_pk>.*)/titolo/',
       'libreria.view.modifica_titolo'),
```

Questo schema, per poter essere richiamato, deve essere inserito prima di quello dell'Admin:

```
(r'^admin/(.*)', admin.site.root))
```

A questo punto, possiamo provare a modificare i titoli direttamente nell'elenco dei libri nel pannello di Admin. Nella Figura 20.2 vediamo alcuni tentativi, uno andato a buon fine e due no: il primo per via del titolo nullo che abbiamo inserito e il secondo perché *Il Profumo* è già presente come titolo in un altro libro (avendo usato l'operatore `titolo__iexact`, il controllo viene fatto ignorando lettere maiuscole e minuscole).

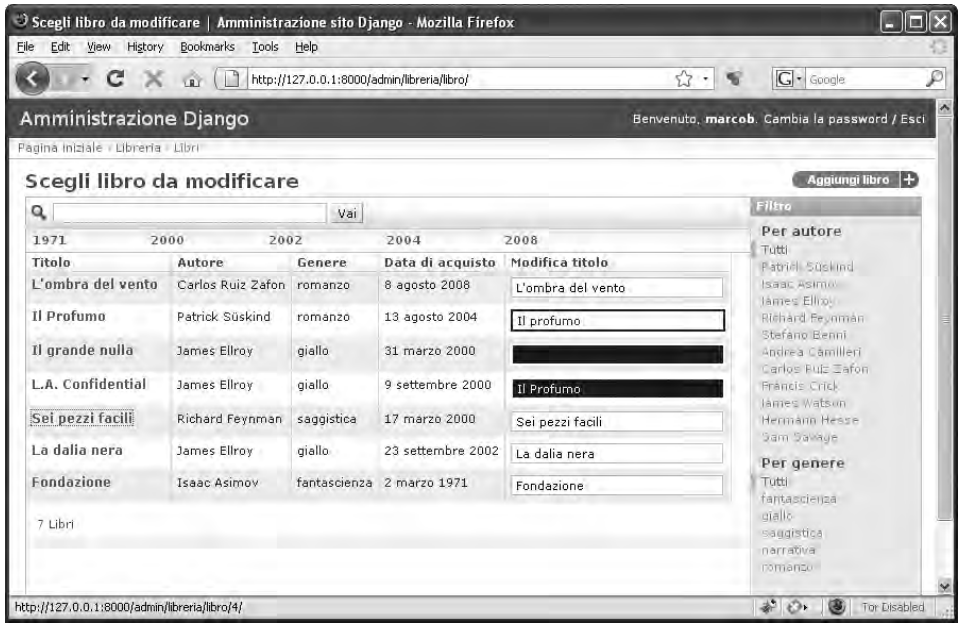

Figura 20.2 Un titolo modificato e due no (uno nullo e uno doppio).

> **MA DOVE SONO I BORDI VERDI E ROSSI NELLA FIGURA 20.2?**
>
> Siccome rosso e verde si confondono nelle immagini in bianco e nero, abbiamo usato degli stili leggermente diversi per segnalare i salvataggi (il bordo spesso) e gli errori (scritta bianca su lo sfondo scuro). Dal vivo sono sicuramente da preferire i bordi verdi e rossi.

Riepilogo

In questa ricetta abbiamo visto come:

- installare jQuery;
- creare la colonna per la modifica;
- visualizzare la colonna;
- creare la funzione Ajax;
- inserire jQuery e la funzione Ajax nel template;
- creare la vista di salvataggio;
- infine, attivare la vista.