02NPYPD - LINGUAGGI E AMBIENTI MULTIMEDIALI A

USER MODELING

Modellazione degli utenti e autenticazione

Scopo di questa esercitazione è quello di apprendere alcuni concetti base per la modellazione in Rails (*Active Record*, *migrations*, ...) e predisporre i meccanismi di autenticazione degli utenti.

WARM UP

Scaricare il progetto base per questa esercitazione presente su GitHub (tramite l'operazione di *clone* o scaricando il file compresso corrispondente).

Come mostrato in aula (e come potete vedere), si è aggiunto qualche contenuto alle pagine già esistenti e un foglio di stile, supportato dal framework Bootstrap sviluppato da Twitter (http://getbootstrap.com). Per includere tale framework nell'applicazione, si è aggiunta la gemma *bootstrap-sass* (versione 3.1.1); pertanto, occorre aggiornare le gemme installate nel progetto con l'operazione *bundle install*.

Osservando il codice del layout generico delle viste (`application.html.erb`), si può notare che è un può "disordinato". Si possono riconoscere tre *unità logiche*: l'header, il footer e l'HTML shim, che dovrebbero essere riunite in un posto a loro dedicato. In Rails, si possono utilizzare i *partial* per far questo, spostando ogni unità logica in un file il cui nome inizia con "_" e sostituendo il loro codice con l'istruzione

```
<%= render "nome-del-partial" %>
```

Otterremo così tre nuovi file: `_shim.html.erb`, `_header.html.erb` e `_footer.html.erb`.

ESERCIZIO 1

In questo primo esercizio affrontiamo i concetti relativi alla modellazione in Rails. Prima di tutto, assicuriamoci di avere una gemma di un DBMS, come sqlite3, nel `gemfile` del nostro nuovo progetto.

Procediamo poi alla creazione del modello dell'utente (`User`) che avrà due attributi, nome ed email, entrambi di tipo `string`:

```
rails generate model User name:string email:string
```

La creazione del modello si può anche fare attraverso il menù *Tools > Run Rails Generator...* di RubyMine, selezionando *model* e inserendo `User name:string email:string` nel campo di "*Generator arguments*". Notate che, a differenza dei nomi utilizzati per generare i controller che sono sempre plurali, i nomi utilizzati per generare un modello sono sempre singolari.

Uno dei risultati del comando di generazione del modello (oltre al modello stesso) è un file chiamato **migration**: le migrazioni forniscono un modo per alterare incrementalmente la struttura di un database, così che il modello possa adattarsi man mano che i suoi requisiti cambiano. Il file in questione si troverà nella cartella `db/migrate` e avrà un nome del tipo `[timestamp]_create_users.rb`.

Per eseguire la migrazione sul database, a questo punto, usiamo il task *db:migrate* di rake:

```
bundle exec rake db:migrate
```

In RubyMine, questo task si può eseguire dal menù *Tools > Run Rake Tasks...*, selezionando *db:migrate.* Tenete presente che quasi tutte le migrazioni sono reversibili utilizzando `db:rollback` come task di `rake`.

Apriamo ora il modello utente creato (`user.rb`) e assicuriamoci che gli attributi nome ed email siano accessibili, cioè che tali attributi possono essere modificati automaticamente da utenti esterni (come quelli che possono sottomettere richieste utilizzando un browser). Pertanto, in `user.rb` dovrà esserci la riga `attr_accessible :name, :email`.

ESERCIZIO 2

Aggiungiamo, insieme, un paio di gemme nel Gemfile (`annotate` e `bcript-ruby`) che ci serviranno nel resto di questa esercitazione. Oltre alle gemme, modifichiamo il file `application.html.erb` per includere il metodo `debug`[1] in modo da poter vedere, direttamente sul sito web, alcune informazioni di debug solo quando l'ambiente utilizzato sarà quello di "development". Il risultato di questa modifica è un riquadro sotto il footer della nostra applicazione web che mostrerà, in formato YAML, informazioni come il controller associato alla pagina visualizzata e il nome dell'azione che ha portato a tale pagina. Modifichiamo anche il foglio di stile per avere il riquadro di debug sul fondo di ogni pagina.

Torniamo ora al modello degli utenti e aggiungiamo qualche elemento di validazione. Vogliamo, infatti, che un utente non possa registrarsi al sito senza inserire il nome e la mail, con un indirizzo mail non valido, oppure che si registri due volte con la stessa mail.

Per far queste validazioni, si può usare il metodo `validates` direttamente nel file `user.rb` che rappresenta il modello. Vogliamo che:

- il nome sia sempre presente (`presence`) e che abbia una lungezza (`length`) massima di 50 caratteri;
- l'indirizzo mail sia sempre presente, sia unica (`uniqueness`) e che abbia un formato (`format`) del tipo *qualcosa@qualcosa.polito.it* (cioè, deve essere ammesso un indirizzo @polito.it e @studenti.polito.it)

Provare a realizzare queste due validazioni, eventualmente aiutandosi con la documentazione del metodo `validates`[2].

ESERCIZIO 3

Rafforziamo il concetto di unicità dell'indirizzo mail presente nel database, creando una nuova migrazione:

```
rails generate migration add_index_to_users_email
```

In RubyMine, tale migrazione si può generare anche utilizzando il menù *Tools > Run Rails Generator...*, selezionando *migration* e inserendo `add_index_to_users_email` nel campo di "*Generator arguments*".

[1] http://api.rubyonrails.org/v3.2.16/classes/ActionView/Helpers/DebugHelper.html#method-i-debug
[2] http://api.rubyonrails.org/v3.2.16/classes/ActiveModel/Validations/ClassMethods.html#method-i-validates

Tale migrazione non è definita, quindi dobbiamo riempire la classe corrispondente con del codice che permetta di aggiungere un indice alla colonna email per assicurare davvero l'unicità dei suoi campi. Tale operazione viene fatta utilizzando il metodo `add_index`[3] con tre parametri: il simbolo corrispondente alla tabella da modificare (`:users`, in questo caso), il simbolo associato alla colonna a cui applicare l'indice (`:email`) e la condizione di unicità (`unique: true`).

A questo punto, possiamo migrare le modifiche al database.

ESERCIZIO 4

Aggiungiamo, infine, la possibilità per un utente di essere riconosciuto sul sito utilizzando una password. Per motivi di sicurezza, però, non memorizzeremo la password in chiaro nel database, ma solo un suo digest, cioè la "versione crittografata" della password stessa. Per farlo, dopo esserci assicurati di avere la gemma `bcrypt-ruby` installata (necessaria per creare la hash della password), procediamo alla creazione di una nuova migrazione:

```
rails generate migration add_password_digest_to_users password_digest:string
```

Eseguiamo ora la migrazione per aggiornare il database con le nostre modifiche.

Ora dobbiamo aggiornare il modello (`user.rb`) in modo da utilizzare la gemma appena aggiunta e in modo da prevedere un campo per la password. Aggiungiamo, innanzitutto, il metodo `has_secure_password`[4] (non ha parametri) alla classe: tale metodo viene messo da Rails e fornisce, essenzialmente, un sistema di autenticazione completo. `has_secure_password` mette a disposizione due attributi "virtuali" (nel senso che non saranno resi persistenti sul DB): `password` e `password_confirmation`. Aggiungiamo le seguenti caratteristiche a tali attributi:

- essi dovranno essere accessibili dall'esterno;
- `password` dovrà avere una lunghezza minima di 8 caratteri.

[3] http://api.rubyonrails.org/v3.2.21/classes/ActiveRecord/ConnectionAdapters/SchemaStatements.html#method-i-add_index

[4] Il metodo has_secure_password realizza l'autenticazione via password che vogliamo ottenere: confronta la password e la sua conferma per verificarne l'uguaglianza, crittografa la password inserita dall'utente, la confronta col digest memorizzato nel DB, aggiunge un metodo chiamato chiamato authenticate che confronta la password crittografata con la password_digest presente nel database, ecc.