Guida Ruby on Rails: architettura di un'applicazione

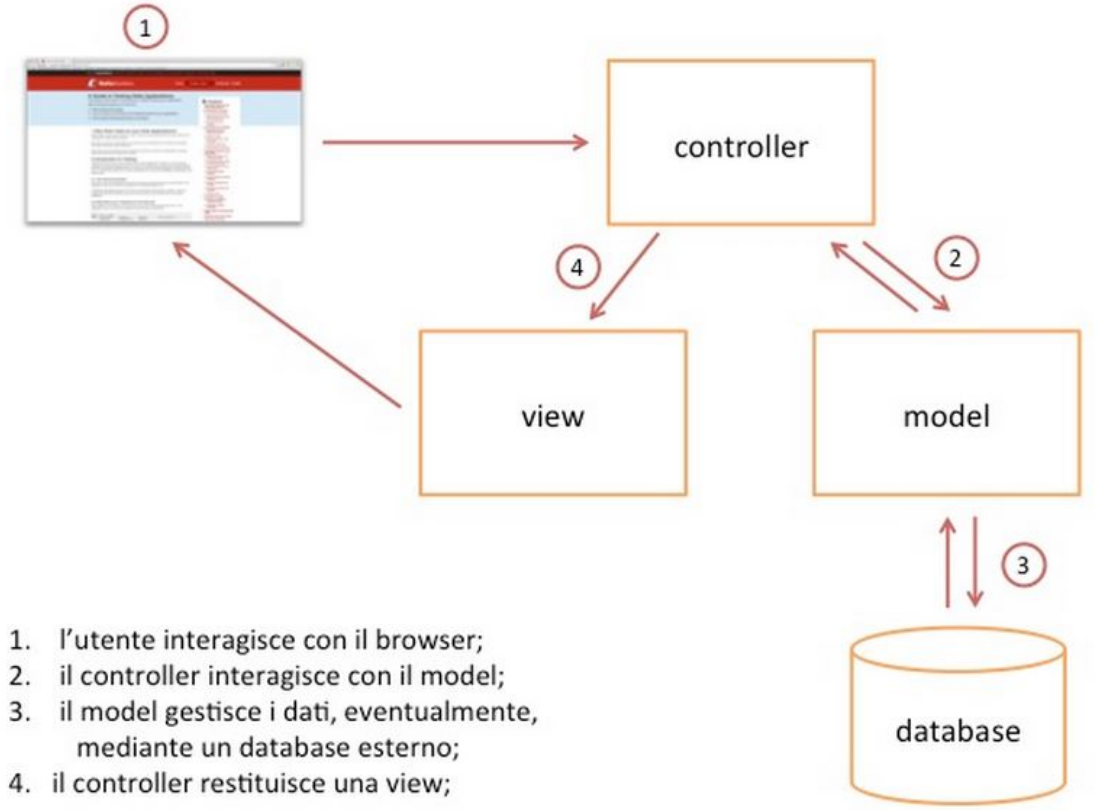
Nel corso di quest’articolo affronteremo la teoria del *design pattern* che governa le applicazioni in Rails, ovvero il già accennato triumvirato Model, View, Control. Capiremo, nello specifico, come sia possibile che un'imposizione strutturale così rigida possa, in realtà, agevolare così tanto il lavoro dello sviluppatore.

Model, View, Controller

Il pattern MVC, ideato dal norvegese Trygve Reenskaug nel 1979, prevede la separazione delle applicazioni in tre componenti principali dai differenti ruoli: il Model, la View e il Controller.

• Il **Model** è responsabile del mantenimento dello stato dell’applicazione, sia di quello provvisoriamente legato alle interazioni dell’utente, sia di quello immagazzinato all’interno di un database. Il Model fornisce alla View e al controller rispettivamente le funzionalità per l’accesso e per l’aggiornamento definendo le regole per l’interazione con i dati. È bene specificare, però che il Model non dipende né dalle View né dai Controller con cui interagisce, ma si limita a fornire metodi che espongano i dati in sola lettura alle View e metodi che permettano di modificare i dati in risposta alle richieste degli utenti ai Controller.
• La **View** si occupa di generare l’interfaccia utente, generalmente basata proprio sui dati del Model, anche se allo stesso Model possono corrispondere numerose View differenti. Ad esempio, in un negozio virtuale i dati relativi agli articoli saranno gestiti dal Model, ma saranno le View a formattarli in maniera definitiva per l’utente finale. La View, però, dopo aver catturato gli input e le scelte dell’utente, delega al Controller il compito di eseguire i processi.
• Il **Controller** svolge un ruolo metaforicamente simile a quello di un direttore d’orchestra. Riceve gli input dall’esterno e dalle azioni dell’utente sulle View, interagisce con i Model e restituisce l’eventuale View modificata.

In sintesi possiamo riassumere una basilare architettura MVC nel seguente schema.



È chiaro, quindi, come sviluppare un'applicazione basata su un design pattern MVC sia molto più semplice e immediato. La divisione dei compiti nelle tre differenti componenti che abbiamo appena elencato, rendono il codice estremamente flessibile ai cambiamenti e il ruolo dello sviluppatore molto meno faticoso, in quando scrivere e mantenere il codice dividendolo in piccole e separate porzioni è decisamente più efficace.

Rapportando quanto detto a un esempio pratico in Rails, immaginiamo che l'utente abbia premuto il pulsante "*aggiungi al carrello*" in un ipotetico sito di vendita online. Il link a cui rimanda il pulsante sarà http://localhost:3000/products?product_id=5. L'applicazione si comporta, secondo i canoni del MVC, nel seguente modo:

1. Il routing riceve la richiesta, contenente il percorso (/products?product_id=5) e il metodo (POST) dal browser e, dato per appurato che *products* rappresenta il nome del controller, mentre *product_id* l'ID del prodotto selezionato, questo trova il Controller denominato *Products* e invoca l'azione create() solitamente associata a POST.
2. Il Controller *Products* invoca il metodo *create()* che cerca il carrello legato all'utente in questione, quale oggetto gestito dal Model e richiede, sempre al Model, di aggiungere il prodotto

con l'ID 5.
3. Il Model interagisce con un eventuale database esterno per manipolare i dati richiesti.
4. Il Controller *Products* indica quale View restituire al browser e quindi all'utente.

Active Record

In un'applicazione RoR, il modello gestire alcune complicazioni logiche. Generalmente, infatti, è chiamato a interagire con dei database di tipo SQL, ovvero Structured Query Language, che per loro natura utilizzano una logica completamente differente da quella di un linguaggio a oggetti quale Rails, utilizzando set di valori invece di dati e operazioni.

Per colmare questo gap strutturale si fa generalmente ricorso alle librerie ORM, ovvero object-relational mapping. Il compito delle librerie ORM è quello di 'mappare' le tabelle di un database SQL in classi. Ad esempio, se un database possiede una tabella products, la nostra applicazione avrà una classe denominata Product. Ogni riga della tabella products corrisponderà a un oggetto della classe Product, così come ogni colonna rappresenterà un attributo dell'oggetto.

In Rails il sistema ORM è rappresentato da Active Record che segue i canoni generici appena descritti: tabelle=classi, righe=oggetti, colonne=attributi. L'unica peculiarità del sistema impiegato in Rails è la quasi assenza di configurazioni, proprio in rispetto al principio *convention over configuration*, espresso nel capitolo introduttivo di questa guida. Vedremo poi, più avanti, come agisce nel dettaglio.

Action Pack

Come avrai capito dalle precedenti spiegazioni, il Controller e le View hanno spesso contatti tra loro. La View riceve dati dal Controller e il Controller riceve eventi dalla View. Per questo motivo, in Rails le View e i Controller sono gestiti da un singolo componente: l'Action Pack. Attenzione a non cadere in errore! Nonostante entrambi vengano gestiti da Action Pack non bisogna pensare che non esista distinzione logica tra i due componenti, anzi! In Rails, fedelmente al pattern MVC, la View resta la responsabile delle pagine visualizzate nel browser, staticamente con l'HTML o dinamicamente con ERB, PHP e JSP, mentre al Controller restano i compiti precedentemente descritti.

Conclusioni

Dopo aver chiarito questioni teoriche, nella prossima lezione inizieremo ad affrontare la sintassi di Ruby.

GUIDA RUBY ON RAILS: INDICE LEZIONI

2) L'ambiente di lavoro e la nostra prima app
3) Un assaggio di dinamicità
4) Architettura di un'applicazione
5) Il linguaggio Ruby - Parte 1
6) Il linguaggio Ruby - Parte 2
7) Creiamo c-Bookcase
8) Convalida e test
9) Ruby on Rails: page layout
10) Guida Ruby on Rails: creiamo il carrello