Ruby on Rails: La prima vera applicazione

Progetto di Sistemi Web

AA 2025/2026

Filippo Poltronieri filippo.poltronieri@unife.it

Edoardo Di Caro edoardo.dicaro@unife.it

Ripasso: Creazione dell'Applicazione

- Creazione automatica della cartella (e delle relative sottocartelle) che conterrà l'applicazione:

```
$ rails new my_app
```

- Avvio del server rails:

```
$ cd my_app
$ bin/rails server
```

Creazione del Model (1)

- Per creare il Model per un'entità del database si utilizza il comando `$ bin/rails generate model` seguito dal nome del model e dagli attributi nella forma **nome:tipo**

```
$ bin/rails generate model Movie title:string duration:integer
```

- ***NB: i nomi dei model devono essere al singolare, in quanto un model instanziato rappresenta un singolo record del database***

Creazione del Model (2)

- Il comando eseguito si occuperà di creare automaticamente:
  - Una migration nella cartella `db/migrate`
  - Un Model ActiveRecords al percorso `app/models/movie.rb`
  - Test e fixture per il modello appena creato

Tipi di dato in Rails (1)

- `:boolean` - Utilizzato per attributi dal valore del tipo "Vero/Falso" o "On/Off".
- `:integer` - Utilizzato per numeri interi (signed, 32 o 64 bit)
- `:bigint` - Simile a `:integer`, ma utilizza un numero di bit variable (quindi può salvare interi estremamente grandi)
- `:float` - Utilizzato per numeri decimali a virgola fissa (fixed-point)
- `:decimal` - Utilizzato per numeri decimali a virgola mobile (floating-point)
- `:string` - Utilizzato per campi di testo brevi, fino a 255 caratteri
- `:text` - Utilizzato per campi di testo lunghi, fino a oltre 30,000 caratteri

Tipi di dato in Rails (2)

- `:date` - Utilizzato per salvare date (anno, mese, giorno)
- `:time` - Utilizzato per salvare tempi (ore, minuti, secondi)
- `:timestamp` - Utilizzato per salvare una data e un tempo
- `:datetime` - Analogo a `:timestamp`
- `:binary` - Utilizzato per salvare immagini, video o audio in formato originale (raw).
- `:references` - Non propriamente un tipo di dato, utilizzato per esplicitare una relazione lato `belongs_to`

Scaffolding in Rails

- In modo molto simile alla creazione del Model, con lo scaffolding in Rails è possibile andare a creare in modo automatico diverse porzioni di codice utile:

```
$ bin/rails generate scaffold Movie title:string duration:integer
```

- Così facendo Rails, tra le altre cose, andrà automaticamente a:
  - Creare il Model
  - Creare il Controller
  - Creare una serie di View (estremamente di base) per la visualizzazione di una lista di elementi, per visualizzare un singolo elemento, per creare un nuovo elemento e per modificare un elemento esistente
  - Creare i file di Test per Model e Controller
  - Aggiornare il file `config/route.rb` per includere le nuove informazioni di routing

La nostra prima applicazione in Rails

Overview dell'Applicazione

- Andiamo a sviluppare la nostra prima applicazione, in modo incrementale
- Seguiremo l’esempio del noto **Internet Movies Database imdb**, andando a creare il nostro my_imdb
- Di cosa dobbiamo occuparci?

1. Modellare le risorse:
    - **movie**
    - **actor**
2. Modellare le relazioni tra **movie** e **actor**: quali sono?
3. In futuro aggiungeremo la possibilità di aggiungere review, autenticarsi e rendere la nostra applicazione reattiva utilizzando turbo-rails

Creazione dell'applicazione

- Creiamo l'applicazione con:
  `$ rails new my_imdb`
- Una volta creata, dobbiamo entrare nel progetto:
  `$ cd my_imdb`
- Poi occupiamoci di creare la risorsa più semplice, actor:
  `$ bin/rails generate scaffold actor name:string dob:date`
- Il generatore scaffolding è il metodo più semplice per generare **automaticamente tutto il codice boilerplate della nostra applicazione**. È la soluzione migliore se vogliamo realizzare un prototipo molto velocemente. Se vogliamo rendere la nostra applicazione più reattiva, potremmo dover modificare il codice generato in seguito.

Output

```
└─$ bin/rails generate scaffold actor name:string dob:date                          invoke  active_record
      create    db/migrate/20250825160031_create_actors.rb
      create    app/models/actor.rb
      invoke    test_unit
      create      test/models/actor_test.rb
      create      test/fixtures/actors.yml
      invoke  resource_route
       route    resources :actors
      invoke  scaffold_controller
      create    app/controllers/actors_controller.rb
      invoke    erb
      create      app/views/actors
      create      app/views/actors/index.html.erb
      create      app/views/actors/edit.html.erb
      create      app/views/actors/show.html.erb
      create      app/views/actors/new.html.erb
      create      app/views/actors/_form.html.erb
      create      app/views/actors/_actor.html.erb
      invoke    resource_route
      invoke    test_unit
      create      test/controllers/actors_controller_test.rb
      create      test/system/actors_test.rb
      invoke    helper
      create      app/helpers/actors_helper.rb
      invoke      test_unit
      invoke    jbuilder
      create      app/views/actors/index.json.jbuilder
      create      app/views/actors/show.json.jbuilder
      create      app/views/actors/_actor.json.jbuilder
```

Controllare le Migration

- Possiamo controllare il file di migration generato, aggiungendo dei valori di default:

  `$ cat db/migrate/20250825160031_create_actors.rb`

```ruby
class CreateActors < ActiveRecord::Migration[8.0]
  def change
    create_table :actors do |t|
      t.string :name, default: "John Doe"
      t.date :dob, default: "1990-09-01"

      t.timestamps
    end
  end
end
```

Eseguire le Migration

- Una volta modificato il file, possiamo procedere.

  `$ bin/rails db:migrate`

```
== 20250825160031 CreateActors: migrating ======================================
-- create_table(:actors)
   -> 0.0023s
== 20250825160031 CreateActors: migrated (0.0023s) =============================
```

La nostra applicazione fin qui (1)

- Lanciamo ora la nostra applicazione e vediamo il layout corrente.
  `$ bin/dev`
- Lo scaffolding ci ha generato tutte le view necessarie (molto semplici) a gestire la risorsa *actor*:
  - **/actors** -> restituisce la lista degli attori presenti (vuota)
  - **/actors/new** -> restituisce il form per la creazione di un nuovo attore.

/actors

Actors

New actor

/actors/new

New actor

Name

Dob

gg/mm/aaaa

Create Actor

Back to actors

Modificare lo stile

- Per ora l'applicazione non definisce un file di stile. Per semplificare lo sviluppo possiamo utilizzare **simple.css** (https://simplecss.org/).
- Possiamo modificare il file `app/views/layouts/application.html.erb`, aggiungendo il link al file di stile.
- `<%= stylesheet_link_tag "https://cdn.simplecss.org/simple.css" %>`

Controller - Actor (1)

- E' sempre bene controllare la struttura dell'applicazione, guardando l'albero di directory per capirne il funzionamento.
- Se controlliamo il Controller della risorsa actor vedremo che tutte le azioni di **show, create, update, edit** e **destroy** sono state definite.
- I metodi generati consentono di ottenere sia risposte in formato HTML che in formato JSON.
- Possiamo estendere i controller per ritornare altri tipi di rappresentazione di risorse.
- **before_action** viene utilizzato per impostare la variabile d'istanza `@actor`

Controller - Actor (2)

```rb
## app/controllers/actors_controllers.rb
class ActorsController < ApplicationController
  before_action :set_actor, only: %i[ show edit update destroy ]

  # GET /actors or /actors.json
  def index
    @actors = Actor.all
  end

  # GET /actors/1 or /actors/1.json
  def show
  end

  # GET /actors/new
  def new
    @actor = Actor.new
  end

  # GET /actors/1/edit
  def edit
  end

  # POST /actors or /actors.json
  def create
    @actor = Actor.new(actor_params)

    respond_to do |format|
      if @actor.save
        format.html { redirect_to @actor, notice: "Actor was successfully created." }
        format.json { render :show, status: :created, location: @actor }
      else
        format.html { render :new, status: :unprocessable_entity }
        format.json { render json: @actor.errors, status: :unprocessable_entity }
      end
    end
  end
```

Controller - Actor (3)

```ruby
  # PATCH/PUT /actors/1 or /actors/1.json
  def update
    respond_to do |format|
      if @actor.update(actor_params)
        format.html { redirect_to @actor, notice: "Actor was successfully updated.", status: :see_other }
        format.json { render :show, status: :ok, location: @actor }
      else
        format.html { render :edit, status: :unprocessable_entity }
        format.json { render json: @actor.errors, status: :unprocessable_entity }
      end
    end
  end

  # DELETE /actors/1 or /actors/1.json
  def destroy
    @actor.destroy!

    respond_to do |format|
      format.html { redirect_to actors_path, notice: "Actor was successfully destroyed.", status: :see_other }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_actor
      @actor = Actor.find(params.expect(:id))
    end

    # Only allow a list of trusted parameters through.
    def actor_params
      params.expect(actor: [ :name, :dob ])
    end
end
```

Esercizi Actor

- Specificare le validation per la risorsa Actor
- Dove vanno indicate?

Movies (1)

- Andiamo ora a modellare la risorsa *movie*
- Adottiamo anche qui un approccio semplificato in cui modelliamo **il titolo, la durata, il/la regista, la trama, e la location principale in cui è stato girato.**
- Anche qui possiamo adottare l'approccio scaffolding per generare model, view e controller con un solo comando

```
$ bin/rails generate scaffold movie title:string director:string
duration:integer plot:text location:string
```

Movies (2)

- Possiamo anche qui aggiungere un valore di default ad alcuni campi come *location* e *plot*.

```ruby
class CreateMovies < ActiveRecord::Migration[8.0]
  def change
    create_table :movies do |t|
      t.string :title
      t.string :director
      t.integer :duration
      t.text :plot, default: "A cool movie"
      t.string :location, default: "Canada"

      t.timestamps
    end
  end
end
```

Movies (3)

- Tutte le routes per la risorsa movie sono state aggiunte!

```
                Prefix Verb   URI Pattern                  Controller#Action
                movies GET    /movies(.:format)            movies#index
                       POST   /movies(.:format)            movies#create
             new_movie GET    /movies/new(.:format)        movies#new
            edit_movie GET    /movies/:id/edit(.:format)   movies#edit
                 movie GET    /movies/:id(.:format)        movies#show
                       PATCH  /movies/:id(.:format)        movies#update
                       PUT    /movies/:id(.:format)        movies#update
                       DELETE /movies/:id(.:format)        movies#destroy
```

- Qual è la root della nostra applicazione?
- Possiamo impostarla attraverso il file di `config/routes.rb`
- Il file generato da rails contiene commenti e indicazioni utili, ci basterà definire la direttiva `root "post#index"`, per impostare la nuova radice del nostro my_imdb.

Cosa ci manca?

- Abbiamo una risorsa `movie` e una risorsa `actor`
- Non abbiamo modellato ancora la relazione `N:M` tra queste due risorse
- Ovviamente un movie avrà associato da 1 a più attori e viceversa un attore reciterà in più film.
- Vogliamo modellare attraverso questa relazione il personaggio interpretato.
- Nelle slide sul model abbiamo visto che ci sono due possibili modi di generare una relazione `N:M`, quali sono? Quale dobbiamo utilizzare in questo caso?

Modellare la Relazione - Generare la Risorsa (1)

- Vogliamo creare un join model `Casting` che realizzi l'associazione tra `actor` e `movie`
- Possiamo utilizzare il generator per creare la risorsa:
  `$ bin/rails generate resource Casting movie:references actor:references character:string`
- Poi dobbiamo aggiornare i model con le rispettive relationship.

Modellare la Relazione - Generare la Risorsa (2)

```
$ bin/rails generate resource Casting movie:references actor:references character:string
      invoke  active_record
      create    db/migrate/20250826083526_create_castings.rb
      create    app/models/casting.rb
      invoke    test_unit
      create      test/models/casting_test.rb
      create      test/fixtures/castings.yml
      invoke  controller
      create    app/controllers/castings_controller.rb
      invoke    erb
      create      app/views/castings
      invoke    test_unit
      create      test/controllers/castings_controller_test.rb
      invoke    helper
      create      app/helpers/castings_helper.rb
      invoke      test_unit
      invoke  resource_route
       route    resources :castings
```

Modellare la Relazione - Migration (1)

- Possiamo vedere il contenuto della migration:

```ruby
class CreateCastings < ActiveRecord::Migration[8.0]
  def change
    create_table :castings do |t|
      t.references :movie, null: false, foreign_key: true
      t.references :actor, null: false, foreign_key: true
      t.string :character

      t.timestamps
    end
  end
end
```

Modellare la Relazione - Migration (2)

- `movie` e `actor` sono indicate come chiavi esterne in `Casting` per modellare la relazione
- Inoltre, è presente l'attributo `reference` che ci consente di modellare il ruolo che l'attore assume nel `movie`
- **Ovviamente, il modello potrebbe essere più complicato in situazioni realistiche** come:
  - un attore che ha più ruoli in uno stesso film
  - la possibilità che anche il regista partecipi al film con un ruolo di attore e così via
  - Tuttavia per lo scopo didattico di questa lezione non andremo ad affrontare queste tematiche.

Next steps (1)

Quali sono i prossimi passi che dobbiamo mettere in atto:

1. Eseguire la migration
2. Modificare i rispettivi model
3. Creare / Modificare le view per l'inserimento dei ruoli nel film

```
bin/rails db:migrate
```

Vanno poi aggiornati i model di `movie` e `actor`

Modificare i Model

```ruby
# app/model/actor.rb
class Actor < ApplicationRecord
  has_many :castings, dependent: :destroy
  has_many :movies, through: :castings
end
```

```ruby
# app/model/movie.rb
class Movie < ApplicationRecord
  has_many :castings
  has_many :actors, through: :castings
end
```

- Come vediamo, `casting` è la join table che collega le due relazioni
- `has_many :movies, through: :castings` e `has_many :actors, through: :castings` indicano che l'associazione con tra le risorse `movie` e `actor` viene effettuata con un HMT

Next steps (2)

Il generator di Rails ha creato anche le rispettive routes per la risorsa casting. Tuttavia potremmo non avere bisogno di tutte queste route.

- L'inserimento di un nuova nuova risorsa `casting` può essere effettuata all'atto della creazione di un `movie`, oppure quando modifichiamo un `movie`.
- Tutte queste azioni vanno gestite sia a livello di controller che a livello di `view`.
- Possiamo iniziare andando a creare qualche attore, in modo da poter poi procedere con la creazione di un `movie` e impostare relativi `casting`.
- Ci sono diversi modi in cui possiamo gestire l'inserimento di attori nel cast, il più semplice potrebbe essere partire della pagina `show` di `movie`

Modificare i Controller

- Il file `castings_controller.rb` è vuoto al momento e vanno specificate le diverse azioni.
- Vogliamo consentire le operazioni di `new`, `create` per l'inserimento di un nuovo attore nel cast
- Rimangono importanti anche le operazioni di `edit`, `update` e `delete` per gestire eventuali errori di inserimento e per rimuovere un personaggio del cast.
- Iniziamo con l'implementare il controller.
- Per prima cosa iniziamo con le operazioni della creazione di un nuovo membro del cast.

Implementazione di `new` e `create`

```ruby
# from app/controllers/castings_controller.rb
class CastingsController < ApplicationController
  before_action :set_movie, only: %i[ new create edit ]
  before_action :set_casting, only: %i[ edit update destroy ]

  def new
    @casting = @movie.castings.new
  end

  def create
    @casting = @movie.castings.new(casting_params)
    if @casting.save
      redirect_to @movie, notice: "Casting was successfully created."
    else
      render :new, status: :unprocessable_entity
    end
  end
```

- In caso di inserimento corretto si viene reindirizzati alla pagina del film

Come implementare le operazioni mancanti?

- Dobbiamo creare il codice per i metodi di `edit`, `update` e `destroy`
- Anche in questo caso le operazioni sono molto semplici, dobbiamo chiamare i rispettivi metodi di `update` sull'istanza di `casting`
- Per il metodo `destroy` l'operazione è invece analoga

Modificare le View (1)

Nella view `show` di movie possiamo aggiungere la visualizzazione del cast (se presente) e chiamare il rendering del form per l'aggiunta di un membro di un cast.

```
# app/views/movies/show.html.erb
<%= render @movie %>

<div>
  <%= link_to "Edit this movie", edit_movie_path(@movie) %>
  |
  <%= link_to "Back to movies", movies_path %>

  <%= button_to "Delete this movie", @movie, method: :delete %>
</div>
```

Modificare le View (2)

Infine vanno create le view per il casting.

- Qui possiamo adottare un processo semplificato, andando a definire il partial `_casting.html.erb` per la visualizzazione delle informazioni del cast
- Una partial view di `_form.html.erb` per l'inserimento del membro

Modificare i Partial (1)

Possiamo implementare i partial in questo modo, per consentire una visualizzazione più efficace.

```
# app/view/castings/_casting.html.erb
<div class="casting">
  <strong><%= casting.actor.name %></strong>
  as
  <em><%= casting.character %></em>
  .
  <%= link_to "Edit", edit_casting_path(casting) %>|
  <%= link_to "Remove",
  casting_path(casting),
  data: {
    turbo_method: :delete,
    turbo_confirm: "Remove this cast member?",
  } %>
</div>
```

Modificare i Partial (2)

Il partial `_form.html.erb` ci consente invece di inserire un nuovo membro del cast, selezionandolo tra gli attori presenti nel database.

```
<%= form_with model: [movie, Casting.new], local: true do |form| %>
  <div class="field">
    <%= form.label :actor_id, "Actor" %>
    <%= form.collection_select :actor_id,
                               Actor.all,
                               :id,
                               :name,
                               prompt: "Select an Actor" %>
  </div>

  <div class="field">
    <%= form.label :character, "character" %>
    <%= form.text_field :character %>
  </div>

  <%= form.submit %>
<% end %>
```

Migliorare la visualizzazione (1)

- Per migliorare l'esperienza utente, possiamo fare il rendering di casting direttamente nella view `_movie.html.erb`
- Inoltre, una `index` view di movie più compatta ci consente di visualizzare i film presenti in un modo più user-friendly.

Migliorare la visualizzazione (2)

```
# app/view/movies/_movie.html.erb
<article id="<%= dom_id movie %>">
  <p>
    <strong>Title:</strong>
    <%= movie.title %>
  </p>

  <p>
    <strong>Director:</strong>
    <%= movie.director %>
  </p>

  <p>
    <strong>Duration:</strong>
    <%= movie.duration %>
    minutes
  </p>

  <p>
    <strong>Plot:</strong>
    <%= movie.plot %>
  </p>

  <p>
    <strong>Location:</strong>
    <%= movie.location %>
  </p>

  <h3>Castings</h3>
  <%= render movie.castings if movie.castings %>
  <%= render "castings/form", movie: movie %>

</article>
```

Migliorare la visualizzazione (3)

```
# app/views/movies/index.html.erb

<% content_for :title, "Movies" %>

<h1>Movies</h1>

<div id="movies">
  <% @movies.each do |movie| %>
    <%= render movie %>
    <div>
      <%= link_to "Edit this movie", edit_movie_path(movie) %>
      |
      <%= button_to "Delete this movie", movie, method: :delete %>
    </div>
  <% end %>
</div>

<%= link_to "New movie", new_movie_path %>
```