Laboratorio con Polymorphic Association

Progetto di Sistemi WEB

Anno Accademico 2025/2026

Filippo Poltronieri

filippo.poltronieri@unife.it

Introduzione

- Nella scorsa lezione, abbiamo aggiunto una colonna `like` al model di `post`. Nel farlo, abbiamo generato una route per l'aggiornamento e la creazione di `like` per il post specifico.
- Abbiamo però detto che non è una soluzione corretta, in quanto non permette di associare il `like` all'utente
- Inoltre, potremmo voler inserire un `like` anche ai commenti dei post.
- Come possiamo fare? Dobbiamo creare un tabella di like sia per post che per i commenti?

Associazione Poliformica

- Rails supporta, tramite ActiveRecord, le associazioni polimorifiche doc.
- Questo tipo di associazone permette di collegare un modello a più modelli tramite una singola associazione.
- Ad esempio, nel nostro Blog, possiamo avere un modello `Like` che può essere associato sia a `Post` che a `Comment`.
- In questo modo, possiamo evitare di creare tabelle separate per ogni tipo di "like".
- Vediamo assieme come fare.

Creazione del modello Like (1)

- Possiamo creare il modello di like utilizzando il generator di Rails

```
rails generate model Like user:references likeable:references{polymorphic}
```

- Questo comando crea una migrazione per la tabella `likes` con una colonna `user_id` e due colonne `likeable_type` e `likeable_id`.
- Cos'è likeable? **Qualche cosa che può essere "liked"**.
- La colonna `likeable_type` conterrà il nome del modello associato (ad esempio, `Post` o `Comment`), mentre `likeable_id` conterrà l'ID dell'istanza specifica di quel modello.
- Controlliamo la `migration` generata.

Creazione del modello Like (2)

```
class CreateLikes < ActiveRecord::Migration[8.1]
  def change
    create_table :likes do |t|
      t.references :user, null: false, foreign_key: true
      t.references :likeable, polymorphic: true, null: false

      t.timestamps
    end
  end
end
```

- Possiamo eseguire la migrazione e controllare il model di like generato.

Creazione del modello Like (3)

- Per semplificare il processo di gestione dei like, possiamo inserire anche un indice univoco per evitare che un utente possa mettere più di un like allo stesso post o commento.
- Questo velocizza anche le query di ricerca dei like.

```
rails generate migration AddUniqueIndexToLikes
```

- Dobbiamo implementare l'indice nella migration:

```
class AddUniqueIndexToLikes < ActiveRecord::Migration[8.1]
  def change
      add_index :likes, [:user_id, :likeable_type, :likeable_id],
            unique: true,
            name: "index_likes_on_user_and_likeable_unique"
  end
end
```

Definizione delle associazioni (1)

Dobbiamo aggiornare il model di `Like` per definire le associazioni polimorfiche. Il model `like` dovrebbe apparire così (*oltre alle validation*):

```
class Like < ApplicationRecord
  belongs_to :user
  belongs_to :likeable, polymorphic: true
end
```

- Vediamo che `belongs_to :likeable, polymorphic: true` indica che il modello `Like` può appartenere a qualsiasi modello
- Ora dobbiamo definire le associazioni nei modelli `Post` e `Comment` per indicare che possono avere molti `like`.

Definizione delle associazioni (2)

- Nel modello `Post`, aggiungiamo:

```
#app/model/post.rb
has_many :likes, as: :likeable, dependent: :destroy
```

- Nel modello di `Comment`, aggiungiamo:

```
#app/model/comment.rb
has_many :likes, as: :likeable, dependent: :destroy
```

- Infine, possiamo specificare l'association anche nel model di `User`, per indicare che un utente può avere molti `like`:

```
#app/model/user.rb
has_many :likes, dependent: :destroy
```

Aggiornamento delle route

- Dobbiamo modificare il file di `config/routes.rb` per gestire le rotte dei `like` in modo polimorfico. In particolare, ci interessa aggiungere le rotte nidificate per `posts` e `comments`.

```
resources :posts do
 resources :comments, only: [:index, :create, :destroy]
# like al post
 resources :likes, only: [:create, :destroy]
end
resources :comments, only: [] do
 resources :likes, only: [:create, :destroy] # like al commento
end
```

Definizione del LikesController

```
class LikesController < ApplicationController
  before_action :set_likeable

 def create
    Like.find_or_create_by!(user: Current.user, likeable: @likeable)
    respond_to do |f|
      f.turbo_stream
      f.html { redirect_back fallback_location: @likeable }
    end
  end

  def destroy
    Like.find_by(user: Current.user, likeable: @likeable)&.destroy
    @likeable.reload.
    respond_to do |f|
      f.turbo_stream
      f.html { redirect_back fallback_location: @likeable }
    end
  end

  private

  def set_likeable
    if params[:post_id]
      @likeable = Post.find(params[:post_id])
    elsif params[:comment_id]
      @likeable = Comment.find(params[:comment_id])
    end
  end
end
```

Definizione delle view

- Qui abbiamo diverse strade, la più semplice è quella di generare un partial `_like.html.erb` che verrà utilizzato sia dalle view di `post` che di `comment`.
- Nella view dobbiamo dare la possibilità di inserire o rimuovere un `like`, *se già inserito*
- Qui dobbiamo gestire *la logica di visualizzazione* con cautela.
- Come facciamo a capire se l'utente ha già *messo like a un post/commento?*
- Dobbiamo caricare i like dell'utente, verificarne la presenza e modificare la view sulla base di questa.

Trovo i like dell'utente

- Per trovare i like dell'utente corrente, possiamo utilizzare `Current.user.likes`.
- Questo ci permette di accedere facilmente ai like associati all'utente attualmente loggato.
- Possiamo quindi verificare se esiste un like per il `likeable` corrente (che può essere un post o un commento) utilizzando `exists?`.
- Tuttavia, in Rails 8, Current.user non è sempre definito, **anche se l'utente non è loggato**, per scelta del Framework doc.

Trovo i like dell'utente

- Infatti, se guardiamo il model di current, vediamo che:

```
class Current < ActiveSupport::CurrentAttributes
  attribute :session
  delegate :user, to: :session, allow_nil: true
end
```

- `Current.user` può essere `nil` se non c'è una sessione attiva.
- Dobbiamo assicurarci che la sessione sia sempre impostata correttamente in ogni richiesta.

Trovo i like dell'utente

- Possiamo modifcare il `ApplicationController` per settare la sessione corrente:

```
class ApplicationController < ActionController::Base
  include Authentication
  before_action :set_current_session
  helper_method :current_user
  # Only allow modern browsers supporting webp images, web push, badges, import maps, CSS nesting, and CSS :has.
  allow_browser versions: :modern
  # Changes to the importmap will invalidate the etag for HTML responses
  stale_when_importmap_changes
  private

  def set_current_session
    Current.session = Session.find_by(id: cookies.signed[:session_id])
  end
  def current_user = Current.user
end
```

- In questo modo, `Current.user` sarà sempre definito (o `nil` se l'utente non è loggato) in ogni richiesta.

Definizione delle view (1)

- Ora possiamo creare il partial `_like_button.html.erb` per gestire il bottone di like/unlike.
- Qui nella view andiamo a controllare se l'utente ha già messo like al `likeable` corrente.
- Se sì, mostriamo il bottone "Unlike", altrimenti mostriamo il bottone "Like".
- Stiamo *invalidando la separazione di MVC*, avrendo logica di controllo nella view, ma in questo caso è accettabile per semplicità.

Definizione delle view (1)

- Concern `likeable.rb` potrebbe essere un'alternativa migliore per gestire questa logica.

```ruby
# app/models/concerns/likeable.rb
module Likeable extend ActiveSupport::Concern
  included do
    has_many :likes, as: :likeable, dependent: :destroy
  end
  def like_for(user)
    likes.detect { |like| like.user_id == user.id }
  end
  def likes_count
    likes.count
  end
end
```

Definizione delle view (2)

```
# app/views/like/_like.html.erb
<%= turbo_stream_from likeable %>

<%= turbo_frame_tag dom_id(likeable, :likes) do %>
  <% user_like = likeable.likes.find_by(user: current_user) %>
  <div class="likes">

    <span id="<%=dom_id(likeable, :likes_count)%>">
      <%= likeable.respond_to?(:likes_count) ? likeable.likes_count : likeable.likes.count %>
    </span>

    <% if user_like %>
      <%= button_to "Unlike",
      polymorphic_path([likeable, user_like]),
      method: :delete,
      class: "btn btn-secondary" %>
    <% else %>
      <%= button_to "Like", polymorphic_path([likeable, :likes]), method: :post, class: "btn btn-primary" %>
    <% end %>
  </div>
<% end %>
```

Definizione delle view (3)

- Dobbiamo gestire il refresh del bottone e del conteggio dei like quando un utente mette o rimuove un like.
- Possiamo definire due partial turbo stream per gestire questi aggiornamenti.

```
# app/views/likes/create.turbo_stream.erb
<%= turbo_stream.replace dom_id(@likeable, :likes) do %>
  <%= render "likes/like", likeable: @likeable %>
<% end %>
```

```
# app/views/likes/destroy.turbo_stream.erb
<%= turbo_stream.replace dom_id(@likeable, :likes) do %>
  <%= render "likes/like", likeable: @likeable %>
<% end %>
```

Model-View: Aggiornamento Like (1)

- Nel model di like, possiamo gestire una callback per aggiornare il conteggio dei like nel modello associato.
- Possiamo gestire l'aggiornamento con *Turbo* e le sue funzionalità di aggiornamento automatico.
- Per farlo abbiamo bisogno di:
  - Una partial per il conteggio dei like `_likes_count.html.erb`
  - Una callback nel model di like per eseguire il broadcast dell'aggiornamento.
- La partial per il conteggio è molto semplice, ci basterà mostrare il conteggio dei like.

```
<%= likeable.likes.count %>
<%= "Like".pluralize(likeable.likes.count) %>
```

- Questa partial andrà a sostituire il contenuto del `span` con ID `likes_count_#{dom_id(likeable)}` ogni volta che viene eseguito il broadcast.

Model-View: Aggiornamento Like (2)

Per gestire l'aggiornamento automatico, possiamo inserire questa callback nel model di `Like`:

```
class Like < ApplicationRecord
### CODICE PRECEDENTE ###
  after_commit :broadcast_like_count, on: [:create, :destroy]

    private
    def broadcast_like_count
      Rails.logger.debug "AFTER_CREATE_COMMIT like##{id} for #{likeable.class}##{likeable_id}"
      Turbo::StreamsChannel.broadcast_replace_to(
      likeable,
      target: ActionView::RecordIdentifier.dom_id(likeable, :likes_count),
      partial: "likes/likes_count",
      locals: { likeable: likeable }
    )
  end
end
```

Post - Comment Views -Aggiornamento

- Dobbiamo includere il partial di like sia nella view di `post` che in quella di `comment`.
- Per consentire l'invio di like ai commenti, dobbiamo assicurarci che il partial venga renderizzato con il commento come `likeable`.
- Per farlo, possiamo semplice inserire l'istruzione di render nel partial di commento.

```
# app/views/comments/_comment.html.erb
<%= render "likes/like", likeable: comment %>
```

```
# app/views/comments/_post.html.erb
<%= render "likes/like", likeable: post %>
```

Gestione Sign-up (Registrazione)

- Dato che Rails authentication non gestisce la registrazione degli utenti, dobbiamo implementare questa funzionalità.
- Possiamo creare un nuovo controller `RegistrationsController` per gestire la registrazione degli utenti.
- Possiamo implementarlo manualmente, guardando come sono gestite le operazioni di `login`` e` logout `nel controller di` SessionsController`.
- Come sempre, dobbiamo ricordarci il nostro pattern **MVC**, e dobbiamo quindi definire:

1. Le rotte per la registrazione
2. Il controller per gestire la registrazione
3. Le view per il form di registrazione
4. Un link alla pagina di registrazione nelle view di login

Gestione Sign-up (Registrazione) - Rotte

- Qui è molto semplice, possiamo modificare il file `config/routes.rb` per aggiungere le rotte di registrazione.

```
 Rails.application.routes.draw do
 # Altre rotte...
 resource :registration, only: [:new, :create]
 end
end
```

- **Attenzione all'uso di `resource` al singolare, dato che ogni utente può avere una sola registrazione**.
- Questo creerà due rotte:
  - `GET /registration/new` per mostrare il form di registrazione
  - `POST /registration` per creare un nuovo utente

Gestione Sign-up (Registrazione) - Controller

- Andiamo a implementare un semplice controller che gestisca le operazioni di `new` e `create`.

```
class RegistrationsController < ApplicationController
  allow_unauthenticated_access
  def new
    @user = User.new
  end
  def create
    @user = User.new(user_params)
    if @user.save
      redirect_to new_session_path, notice: "Account created successfully. Please log in."
    else
      render :new, status: :unprocessable_entity
    end
  end
  private
  def user_params
    puts params.inspect
    params.require(:user).permit(:email_address, :username, :password, :password_confirmation)
  end
end
```

Gestione Sign-up (Registrazione) - View

- Non ci manca che creare la view per il form di registrazione, qui ci basta un semplice form, possiamo prendere spunto da `app/views/sessions/new.html.erb`.
- Di quali campi abbiamo bisogno? Dipende, da come abbiamo definito il nostro `user`, in questo caso gestiamo semplicemente `username`, `email_address`, `password` e `password_confirmation`.

```
<h1>Registration</h1>
<%= form_with model: @user, url: registration_url do |form| %>
  <div>
    <%= form.label :username %>
    <%= form.text_field :username,
                    required: true,
                    autofocus: true,
                    autocomplete: "Username" %>
  </div>
  <div>
    <%= form.label :email_address %>
    <%= form.email_field :email_address, required: true, autocomplete: "email" %>
  </div>
  <div>
    <%= form.label :password %>
    <%= form.password_field :password, required: true, autocomplete: "new-password" %>
  </div>

  <div>
    <%= form.label :password_confirmation %>
    <%= form.password_field :password_confirmation,
                        required: true,
                        autocomplete: "new-password" %>
  </div>

  <div>
    <%= form.submit "Sign up" %>
  </div>
<% end %>
```