AUTORIZZAZIONE

Marco Console
console@dis.uniroma1.it

LABORATORIO

NOTA PRELIMINARE

- Il seguente materiale didattico è basato su un insieme di lucidi prepato dal **Prof. Leonardo Querzoni.**

- Ringrazio il **Prof. Leonardo Querzoni** per avermi concesso di usare il materiale da lui prodotto.

PROGRAMMA DI OGGI

- **Vedremo due gemme per l'autorizzazione**

- **Canard**: Role-based Authorization

- **Cancancan**: Attribute-based Authorization

QUALCHE LINK

- Canard.
  - https://github.com/james2m/canard
- Cancancan:
  - https://github.com/CanCanCommunity/cancancan

A questi link è possibile trovare il codice delle gemme insieme alla documentazione.

CANARD

CANARD

- Una gemma che implementa una forma semplice di RBAC
  - $RBAC_0$ = subjects+roles+resources
  - $RBAC_1$ = gerarchie di ruoli (in una forma rudimentale)

- Basata su CanCanCan
  - Un gemma per autorizzazione che garantisce grande flessibilità
  - A sua volta una evoluzione di CanCan

- Canard semplifica la definizione dei ruoli a scapito della flessibilità.

- Info e Codice: https://github.com/james2m/canard

CANARD E RUOLI

- Con Canard possiamo specificare ruoli diversi per ogni singolo utente
  - $RBAC_0$
- Per farlo dobbiamo modificare il Model che stiamo utilizzando per rappresentare gli utenti nella nostra applicazione
- Per quanto segue assumiamo Users, ma il tutto si applica per qualunque modello state usando
  - Potendo definire ruoli diversi, non c'è bisogno di avere più di un modello per l'autenticazione
  - In caso di più di un modello, è possibile comunque utilizzare Canard con qualche accortezza aggiuntiva. **Sconsigliato.**

ROLE MASKS

- Il Model usato per rappresentare gli utenti deve dichiarare la lista dei ruoli possibili

```
acts_as_user roles => [:role1, … :roleN]
```

- Il Model usato per gli utenti deve avere un campo roles_mask di tipo integer

```
rails g migration add_roles_mask_to_users roles_mask:integer
```

- Il campo **roles_mask** è una maschera di bit che rappresenta i ruoli degli utenti
  - Bit m a 0 => l'utente **NON** ha il ruolo :roleM
  - Bit m a 1 => l'utente **ha** il ruolo :roleM

BIT MASKS

- Un intero può essere visto come una stringa di bit
  - 1 => 00000001
  - 2 => 00000010
  - 3 => 00000011
- Per le maschere di bit dobbiamo usare bitwise operators.
  - Operatore **OR**. Simbolo: |. Esempio: (0 | 2) = 2
    - 00000000b | 00000010b = 00000010b = 2d
  - Operatore **AND**. Simbolo: &. Esempio: (1 & 2) = 0
    - 00000001b | 00000010b = 00000010b = 0d

- Per gestire le maschere di bit nei modelli sarebbe meglio creare dei metodi ad hoc.

GESTIRE ROLES_MASK

- Alcuni metodi utili da implementare
- Assumiamo K essere l'intero con l'M-esimo bit a 1
  - Ex: M = 3 K = 00000100b = 4d

```
Def set_roleM do
   self.roles_mask = self.roles_mask | (K)
   self.save
end
```

```
Def is_roleM? do
   self.roles_mask & (K) = K
end
```

GESTIRE ROLES_MASK

- Assumiamo di avere in totale n ruoli
- Assumiamo K essere l'intero con l'M-esimo bit a 1
- Assumiamo K^{inv}_{n} essere l'inverso di K **troncato a n**
  - Ex: K = 00000100b K^{inv}_{4} = 00001011 = 19d

```
Def unset_roleM do
   self.roles_mask = self.roles_mask & (K^inv_n)
   self.save
end
```

ABILITIES

- Le **ability** definiscono cosa può e non può fare ogni ruolo.
- Una abilità garantisce o nega una certa azione su una risorsa

```
can <Actions>, <Resources>
```

```
cannot <Actions>, <Resources>
```

- Le ability sono in appositi ability file, uno per modello utente.
  - **`app/ability/xxx.rb`**

- Ogni ruolo deve avere un ability file associato
  - Se non ce l'ha, è considerato **cannot** su ogni azione e ogni risorsa

ACTIONS E RESOURCES

- Le **action** vengono indicate con dei symbol.
  - Possiamo aggiungere qualunque symbol vogliamo.

`:read` `:create` `:eat`

- Le **resource** sono nomi delle modelli a cui l’abilità si riferisce.
  - Non sono nomi arbitrari e iniziano con la lettera maiuscola.

`Moviegoer` `Movie` `Review`

ABILITY GENERATOR

- Gli ability files possono essere generati automaticamente

```
rails g canard:ability <Role> <Abilities>
```

- **<Role>** è il ruolo per cui stiamo definendo le abilità

- **<Abilities>** è una lista di abilità con la seguente sintassi.

```
<can|cannot>:<actionName>:<resource>
```

```
<can|cannot>:[<actionList>]:[<resourceList>]
```

ABILITY FILE E GENERATORE

```ruby
Canard::Abilities.for(:Moviegoer) do

  can [:read, :update], Movie
  cannot [:destroy], Movie
  can [:read, :update, :destroy], Review

end

```

```
rails g canard:ability moviegoer
can:[create,read]:[Movies,Reviews]
can:destroy:Review cannot:destroy:Movie
```

HELPER METHODS

- Canard usa gli stessi helper method di CanCanCan

- Per impedire l'accesso a una risorsa

```
authorize! <Action>, <resource>, <params>
```

Lancia un'eccezione se l'utente corrente non è autorizzato, bloccando il flusso dell'esecuzione.

- Per controllare le autorizzazioni:

```
can? <Action>, <resource>
```

ritorna **`true`** se l'utente corrente può eseguire **`<Action>`** su **`<resource>`**.

HELPER METHODS

- Il file `app/controllers/application_controller.rb` deve definire altri due metodi.

```
helper_method :current_user
def current_user
  return <CurrentUserModelInstance>
end
```

- Usato da CanCanCan per ottenere l'utente corrente
- **Va ridefinito se il modello è diverso da User**

```
rescue_from CanCan::AccessDenied do |exception|
  <RispostaAllaViolazione>
end
```

- Rescue method per l'eccezione lanciata da `authorize!`

ESEMPI CANARD

ESEMPIO

- Rails 6.x
  - Canard: https://github.com/james2m/canard
  - Gemfile: **gem 'canard', '~> 0.6.2.pre'**

- Codice da cui partire:
  - `git clone` **https://gitlab.com/console.marco/spoiled-potatoes.git**
  - `git checkout base-autorizzazione`

- Definiamo un ruolo moviegoer per **`User`**
  - Può creare, distruggere, modificare review
  - Può creare Movie
  - Non può modificare o cancellare Movie

ESEMPIO -- DESIGN

- Ruoli
  - Un unico ruolo (per ora). Moviegoer
- Risorse
  - Review e Movie
- Azioni
  - Create, Update, Delete

ESEMPIO -- SET UP

- In Gemfile aggiungiamo

```
gem 'canard', '~> 0.6.2.pre'
```

- Installiamo le gemme

```
bundle install
```

ESEMPIO – SET UP

- In app/controllers/application_controller.rb

```
rescue_from CanCan::AccessDenied do |exception|
  redirect_to root_path, :alert => exception.message
end
```

- Aggiungiamo **roles_mask** a users

```
rails g migration add_roles_mask_to_users
roles_mask:integer
```

```
rake db:migrate
```

USER MODEL

- Aggiungiamo la lista di ruoli….

```
acts_as_user :roles => [:moviegoer]
```

- E qualche helper…

```
def is_moviegoer?
  return (self.roles_mask & 1) == 1
 end
```

```
def set_moviegoer
  self.roles_mask = (self.roles_mask | 1)
  self.save
 end
```

```
def unset_moviegoer
  self.roles_mask = 0
  self.save
 end
```

ESEMPIO – AUTORIZZAZIONE

- Generiamo le abilità.

```
rails g canard:ability moviegoer
can:[create,destroy,update]: Review
can:[create]: Movie
cannot:[read,update]: Movie
```

- Blocchiamo le azioni nei controllers di **`Movie`** e **`Review`**

```
authorize! :action, Resource,
message => "You are not authorized"
```

ESEMPIO – VISUALIZZAZIONE

- Impediamo di visualizzare «Delete» se non abbiamo l'autorizzazione ad utilizzarlo

```
<% if can? :destroy, Movie %>
   <td><%= link_to "Destroy", movie_path(movie.id), method: :delete,
                    data: { confirm: "Are you sure?" } %></td>
<% else %>
<td>Destroy</td>
<% end %>
```

```
<% if not current_user.nil? %>
<b>Roles : </b>
       <% if current_user.is_moviegoer? %>
          Moviegoer
       <%end %>
<% end  %>
```

ESEMPIO – NUOVI UTENTI

- I nuovi utenti devono essere moviegoers
- Estendiamo il controller per le registrazioni
  - app/controllers/users/registration_s_controller.rb

```
class Users::RegistrationsController < Devise::RegistrationsController
   after_action :assign_role, only:[:create]

   def assign_role
      if not current_user.nil?
         current_user.set_moviegoer
      end
   end
end
```

- Aggiorniamo la route per la registrazione

```
devise_for :users, controllers: { registrations:
                  "users/registrations" }
```

ESERCIZIO -- CANARD

- A partire dal codice già scritto.
  - Oppure `git checkout base-autorizzazione-2`

- **Aggiungere un nuovo ruolo (Admin)**
  - **Può cancellare e editare tutto.**
  - **Può cambiare i ruoli degli utenti**
  - **Può impedire agli utenti di compiere qualunque azione (ban)**

ESERCIZIO -- CANARD

- Una soluzione è presente nel seguente branch:
  - **`feature_canard-auth`**

ESERCIZIO – ABILITÀ

- Generiamo le abilità.

```
rails g canard:ability admin
can:[create,read,destroy]:[Movies,Reviews]
```

USER MODEL

- Aggiungiamo la lista di ruoli….

```
acts_as_user :roles => [:moviegoer, :admin]
```

- E qualche helper…

```
def is_admin?
  return (self.roles_mask & 2) == 2
 end
```

```
def set_admin
  self.roles_mask = (self.roles_mask | 2)
  self.save
 end
```

```
def unset_admin
  self.roles_mask = (self.roles_mask & 1)
  self.save
 end
```

ESERCIZIO – AZIONI DELL'ADMIN

- Aggiungiamo routes per le azioni dell'amministratore
  - Vista per la lista utenti
  - Quattro route per le azione (ban\unban, admin\unadmin)

- Aggiungiamo un controller per le azioni
  - app/controllers/admins_controller.rb

- Aggiungiamo una vista per interagire col controller
  - app/views/admins/index.html.erb

- **Dobbiamo proteggere le azioni dell'admin!**

CANCANCAN

ABILITY GENERATOR

- CanCanCan è una gemma più complessa ma si basa sugli stessi concetti che appena visti (ereditati da CanCan)
  - Resource
  - Action
  - User
  - Ability

- La differenza è il modo in cui possiamo definire le abilità.
  - Tramite la classe **`Ability`**
  - **`Ability`** definisce le abilità di ogni utente nel metodo **`initialize`**

- La grande flessibilità di Cancancan ci permette di implementare Atribute Based Access Control

ABILITY GENERATOR

- Nel metodo inizialize, è possibile dichiarare abilità

```
can:<actionName>:<resource>
```

- Il «soggetto» è l'utente corrente.
  - Possiamo definire programmaticamente delle specifiche condizioni per garantire o negare l'autorizzazione su una risorsa.
  - Queste condizioni possono utilizzare attributi dell'utente e condizioni esterne

- La classe ability può essere generata come segue

```
rails g cancan:ability
```

HELPERS

- Il file **app/controllers/application_controller.rb** deve definire altri due metodi.

```
helper_method :current_user
def current_user
  return <CurrentUserModelInstance>
end
```

- Dobiamo riderinire il metodo se usiamo un altro modello.

```
rescue_from CanCan::AccessDenied do |exception|
  <RispostaAllaViolazione>
end
```

- Rescue method per l'eccezione lanciata da **authorize!** In caso di permesso negato (lo stesso di Canard)

ABILITY CLASS

```ruby
class Ability
  include CanCan::Ability

  def initialize(user)

    can [:read,:create], :all

    if user.isCool? and Time.now.hour == 0
        can :destroy, :all
    end
  end
```

ESEMPIO

- Rails 6.x
  - Cancancan: https://github.com/james2m/canard
  - Gemfile: **gem ‘cancancan'**

- Codice da cui partire:
  - `git clone` **https://gitlab.com/console.marco/spoiled-potatoes.git**
  - `git checkout base-autorizzazione`

- Definiamo un attributo **`coolness`** per gli user
  - Utenti con **`coolness`** > 5 possono editare i film
  - Utenti con **`coolness`** >9 possono cancellare i film

ESEMPIO – SET UP

- In Gemfile aggiungiamo

```
gem 'cancancan'
```

- Installiamo le gemme

```
bundle install
```

ESEMPIO – SET UP

- In app/controllers/application_controller.rb

```
rescue_from CanCan::AccessDenied do |exception|
  redirect_to root_path, :alert => exception.message
end
```

ESEMPIO – COOLNESS

- Generiamo un nuovo attributo per Users

```
rails g migration AddCoolnessToUsers coolness:int
```

- Invochiamo la migrazione

```
rake db:migrate
```

ESEMPIO – ABILITÀ

- **Generiamo una ability class con cancancan**

```
rails g cancan:ability
```

- Definiamo le abilità in riferimento all'attributo **`coolenss`**
  - Nella classe ability generata

ESEMPIO – ABILITÀ

```
class Ability
  include CanCan::Ability
  def initialize(user)

  can [:create], :all
  can [:update, :destroy], Review
   if not user.coolness.nil?
    if user.coolness > 5
        can :update, Movie
    end

    if user.coolness > 10
            can :destroy, Movie
        end
    end
  end
end
```

ESEMPIO – ABILITÀ

```
class Ability
  include CanCan::Ability
  def initialize(user)

  can [:create], :all
  can [:update, :destroy], Review
   if not user.coolness.nil?
    if user.coolness > 5
        can :update, Movie
    end

    if user.coolness > 10 and Time.now.min >= 50
            can :destroy, Movie
        end
    end
  end
end
```

ESEMPIO – AUTORIZZAZIONE

- Blocchiamo le azioni nei controllers di **`Movie`** e **`Review`**

```
authorize! :create, Movie,
message => “You are not authorized”
```

- Blocchiamo la visualizzazione

```
<% if can? :destroy, Movie %>
   <td><%= link_to "Destroy", movie_path(movie.id), method: :delete,
                    data: { confirm: "Are you sure?" } %></td>
<% else %>
<td>Destroy</td>
<% end %>
```

ESEMPIO – UN ADMIN

- Come definiamo l'attributo **`coolness`**?
  - Per ora dalla console.

```
$ rails console

- u = User.where(:email => "marco@email.com").first

- u.coolness = 999

- m.save
```

- Dovremmo avere una funzionalità nella logica applicativa

ESERCIZIO -- CANCANCAN

- A partire dal codice già scritto.
  - `O fate checkout del branch feature_cancancan-auth`
- Implementare un controllo di accesso basato sull'indirizzo email associato all'account
  - Solo gli utenti con email @overpower.op possono editare e distruggere Movie