

LABORATOIRE – 5



EN VOITURE POUR RAILS

PARTIE IV

- AUTHENTICATION ET CONTRÔLE DES DROITS
-

1	AUTHENTIFICATION AVEC DEVISE.....	3
2	COMPLEMENTS DEVISE.....	11
2.1	Choix de l'identifiant.....	11
2.2	Configurer les options de Devise.....	11
3	AJOUTER DES RÔLES.....	13
3.1	Ajouter des rôles	13
3.2	Mise en place des relations entre rôles les utilisateurs	13
4	CANCANCAN.....	18
4.1	La définition des droits dans ability.rb	18
4.2	Utiliser les droits pour contrôler l'accès aux actions	20
4.3	Utiliser les droits dans le code des vues	21
4.4	Utiliser les droits pour contrôler l'accès à l'ensemble des actions RestFULL.....	23
5	ANNEXE - DOCUMENTATION CANCANCAN.....	26
5.1	Installation de CanCanCan.....	26
5.2	Comment démarrer	26
5.2.1	Définir les permissions	27
5.2.2	La méthode can	28
5.2.3	Conditions supplémentaires	29
5.2.4	Combinaison des permissions	30
5.2.5	La méthode cannot	30
5.3	Définir des permissions avec des blocs	30
5.4	Contrôler les permissions	30
5.5	Tester les permissions autrement (Functional testing)	31
5.6	Autoriser les actions du contrôleur	32
5.7	Exception handling	35
5.8	Changer les valeurs par défaut	36

Préambule


Dans ce tutoriel nous mettrons en place une petite application sommaire qui implémentera un système d'authentification s'appuyant sur la gem "Devise" et un système d'autorisation, - gestion des droits de chaque utilisateur -, s'appuyant sur la gem "CanCanCan".

1 Authentification avec Devise


Pour illustrer la technique d'authentification, nous allons créer une application rails s'appuyant sur une table `Users`, susceptible d'enregistrer tous les utilisateurs du système avec leur mot de passe.

Les fonctionnalités mises à disposition par **Devise** nous permettrons d'opérer l'enregistrement de l'utilisateur avec spécification de son mot de passe, sa connexion (login) et sa déconnexion (logout). En outre, on verra que **Devise** met à disposition un certain nombre de méthodes comme la méthode `current_user` qui permettra à tout instant de connaître l'utilisateur courant, et donc plus tard, grâce au plugin **CanCanCan**, de connaître les droits de l'utilisateur courant.

■ Création de l'application & installation des librairies

 Commençons par créer une application.


```
> rails new tuto_securite -d mysql
```

 Créer une base de données (le nom que vous désirez) et éditer le fichier `database.yml` en conséquence.


 Installation de **Devise** et **CanCanCan**.

Dans le fichier **gemfile** de notre application ajoutons les gems de "Devise" et de "CanCan".

```
gem 'devise'  
gem 'cancancan'
```

 Effectuer un "bundle install" (en étant placé dans le répertoire «tuto_securite»)

```
>bundle install
```

 Contrôlez l'installation de Devise et CanCanCan

```
C:\railsapp\tuto_secu>gem list devise

*** LOCAL GEMS ***

devise (3.5.6)

C:\railsapp\tuto_secu>gem list cancancan

*** LOCAL GEMS ***

cancancan (1.13.1)
```



Installer le système d'authentification **Devise** dans l'application elle-même.

Depuis le répertoire de l'application entrez la commande:

```
>rails generate devise:install
```

Vous devriez obtenir les informations suivantes:

```
C:\railsapp\tuto_secu>rails generate devise:install
  create  config/initializers/devise.rb
  create  config/locales/devise.en.yml
=====
Some setup you must do manually if you haven't yet:

  1. Ensure you have defined default url options in your environments files. Here
  is an example of default_url_options appropriate for a development environment
  in config/environments/development.rb:

      config.action_mailer.default_url_options = { host: 'localhost', port: 3000 }

  In production, :host should be set to the actual host of your application.

  2. Ensure you have defined root_url to *something* in your config/routes.rb.
  For example:

      root to: "home#index"

  3. Ensure you have flash messages in app/views/layouts/application.html.erb.
  For example:

      <p class="notice"><%= notice %></p>
      <p class="alert"><%= alert %></p>

  4. If you are deploying on Heroku with Rails 3.2 only, you may want to set:

      config.assets.initialize_on_precompile = false

  On config/application.rb forcing your application to not access the DB
  or load models when precompiling your assets.

  5. You can copy Devise views (for customization) to your app by running:

      rails g devise:views
=====
```

■ Mettre en place une gestion des utilisateurs

Dans cette partie du tutorial, nous allons générer un échafaudage (contrôleur, modèle et vues) s'appuyant sur une table « **users** » qui permettra d'enregistrer tous les utilisateurs avec leur mot de passe et diverses informations.



Générons tout d'abord l'échafaudage. Le modèle `User` aura deux attributs : `nom` et `prenom`

```
>rails g scaffold User nom:string prenom:string
```



Générons ensuite par-dessus les champs nécessaires à l'authentification grâce au générateur mis à disposition par Devise.

```
>rails generate devise NomModele
```

Ainsi, saisissez la commande ci-dessous depuis le dossier racine de l'application:

```
>rails generate devise User
```

Vous devriez obtenir les messages suivants, témoignant d'une mise à jour des fichiers `route.rb` et `user.rb`, et de la création d'un fichier de migration:

```
C:\railsapp\tuto_secu>rails generate devise User
  invoke  active_record
  create   db/migrate/20160404112904_add_devise_to_users.rb
  insert   app/models/user.rb
  route    devise_for :users
```



Exécuter la migration pour créer la table `users`



En fait, deux fichiers de migration ont été créés !!

Le premier, **`xxx_create_users.rb`**, a été créé au moment de la génération du scaffold. Il permet de créer une table `users` avec deux colonnes : `nom` et `prenom`.

Le deuxième, **`xxx_add_devise_to_users.rb`**, change la table `users` pour lui ajouter un certain nombre de colonnes, à commencer par le mot de passe (`encrypted_password`).

```
>rake db:migrate
```

Visualisez la table `users` résultant de ces deux migrations :

Column	Type
id	int(11)
nom	varchar(255)
prenom	varchar(255)
created_at	datetime
updated_at	datetime
email	varchar(255)
encrypted_password	varchar(255)
reset_password_token	varchar(255)
reset_password_sent_at	datetime
remember_created_at	datetime
sign_in_count	int(11)
current_sign_in_at	datetime
last_sign_in_at	datetime
current_sign_in_ip	varchar(255)
last_sign_in_ip	varchar(255)

Champs ajoutés par Devise



Par défaut, le champ '**email**' est utilisé comme « login » au moment de l'authentification.



Nous allons exécuter une migration pour rajouter un champ "Telephone" à notre table `Users`.

Par convention (c'est l'usage mais rien ne nous y oblige..), nous donnerons le nom `AddXXXTToYYY` ou `add_xxx_to_yyy` au fichier de migration correspondant.

Avec :

- XXX : Le nom de l'attribut à ajouter
- YYY : Le nom de la table

```
> rails generate migration AddXXXTToYYY attribut:type_attribut
```

ou :

```
> rails generate migration add_xxx_to_yyy attribut:type_attribut
```

Pour ajouter le champ "Telephone" nous entrons donc la commande suivante:

```
> rails generate migration AddTelephoneToUsers telephone:string
```

Cette migration nous a généré un fichier de migration présent dans `/db/migrate` sous le nom `YYYYMMDDHHMMSS_nom_de_la_migration.rb`

Voici le contenu du fichier..

```
class AddTelephoneToUsers < ActiveRecord::Migration
  def change
    add_column :users, :telephone, :string
  end
end
```

Avec l'instruction `add_column` pour l'ajout d'un attribut.

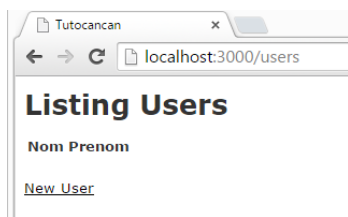
👉 N'oubliez pas le **`rake db:migrate`** pour mettre la base à jour avec la nouvelle colonne.

■ Configuration de Devise

Notre application devrait maintenant être fonctionnelle, testons là un peu!

- Si nous allons voir à l'adresse: <http://localhost:3000/users>

Nous arrivons tout normalement sur l'index des utilisateurs, aucune authentification n'a été requise.



- Si nous essayons maintenant: http://localhost:3000/users/sign_in

Nous arrivons cette fois sur une page d'authentification (page **Sign in**) avec un lien sur « **Sign up** » qui permet d'enregistrer un nouveau compte utilisateur.

Sign in

Email

Password

☐ Remember me

[Sign up](#)

[Forgot your password?](#)

Sign up

Email

Password

Password confirmation

[Sign in](#)

[Forgot your password?](#)



Nous allons forcer l'utilisateur à s'authentifier. Pour se faire il faut rajouter l'option suivante au début du contrôleur `UserController` :

`before_filter :authenticate_user!`

Ce faisant, l'exécution de toute action du contrôleur `UserController` (`index`, `show`, etc..) commencera par invoquer la méthode **`authenticate_user`** (une méthode de `Devise`). Cette méthode contrôle que l'utilisateur courant est correctement authentifié. Si ça

n'est pas le cas, elle redirige sur le formulaire d'authentification. L'idée générale étant par la suite que tout contrôleur dispose d'un tel filtre d'entrée.



L'instruction générique est

```
"before_filter: authenticate_NsomDeLaClasseUtilisateur!"
```



Essayons à nouveau !

=> Cette fois-ci "<http://localhost:3000/users>" nous redirige vers

["http://localhost:3000/users/sign_in"](http://localhost:3000/users/sign_in) et force ainsi l'authentification.



N'allez pas plus loin dans le test, patience !!



Si l'on regarde dans notre répertoire Views du contrôleur « Users », nous ne verrons pas les vues « sign_in »,... elles sont gérées en interne par **Devise**. Si l'on veut modifier l'une d'entre elles, nous devons demander à **Devise** de nous les rendre « public » avec la commande suivante:

```
>rails generate devise:views
```

Dès cet instant, toutes ces vues sont présentes dans le dossier `/app/views/devise`.



Allons modifier la vue d'entête qui apparaîtra une fois que nous serons correctement authentifié!

Devise met plusieurs helpers à disposition, notamment:

- **user_signed_in?** permet de savoir si un utilisateur est authentifié.
- **current_user** contient l'utilisateur actuellement authentifié.

Essayons d'en tirer parti en écrivant ce petit bout de code dans le fichier **application.html.erb** présent dans `/app/views/layout`.

Ce code (il s'agit de la vue d'entête) est à insérer avant l'instruction **<%yield%>**.

```
<div id='user_nav'>
  <% if user_signed_in? %>
    Vous êtes connecté en tant que <%= current_user.email %>
    <%= link_to "Déconnexion", destroy_user_session_path, :method => :delete
  %>
  <%= link_to "Modifiez vos données", edit_user_registration_path %>
  <% elsif request.env['PATH_INFO'] == '/users/sign_in' || \
    request.env['PATH_INFO'] == '/users/sign_up' %>
    <%= link_to "Connexion", new_user_session_path %> ou
    <%= link_to "Enregistrez-vous", new_user_registration_path %>
  <%>
```



```
<% end %>
</div>
```

L'insertion de ce code permet:

- A un utilisateur de voir sous quel compte il est connecté grâce à `current_user.email`.
- A un utilisateur connecté de se déconnecter grâce à `destroy_user_session_path`.
- A un utilisateur connecté de modifier ses données grâce à `edit_user_registration_path`.
- A un utilisateur de naviguer entre les pages `sign_in` et `sign_up` sans devoir passer par l'URL

➡ Occupons-nous encore d'aller modifier la page d'accueil dans `routes.rb` avec l'instruction

```
root 'users#index'
```

Et voici, à peu de chose près, ce à quoi l'on doit arriver après avoir opéré ces différentes modifications :

➡ Enregistrez-vous.. (formulaire sign-up) en signalant votre email et votre mot de passe

Vous obtenez alors l'écran suivant, avec la vue d'entête spécifiée précédemment par nos soins.

■ Analyse de la page « Listing Users »



Notez qu'en l'état, le « sign-up » d'un nouvel utilisateur n'a eu aucun effet sur la liste des utilisateurs, qui apparait vide à l'écran.

En effet, le formulaire sign-up proposé par Devise nous a demandé l'email et le mot de passe. Rien concernant le nom, prénom et téléphone.

Le lien permettant de créer un nouvel utilisateur (New User) sera plus tard supprimé. L'ajout d'un nouvel utilisateur se fera uniquement à l'occasion du « sign-up » de l'utilisateur en question.

Mais on peut d'ores et déjà adapter certaines vues « Users » pour que le téléphone et l'email soient affichés à l'occasion d'un index et d'un show, et mis à jour à l'occasion d'un edit.



Complétez dans ce sens les vues « Users » : `_form.html.erb`, `index.html.erb`, `show.html.erb`.

N'oubliez pas non plus de compléter la liste des paramètres autorisés du contrôleur « users » avec les paramètres `telephone` et `email`:

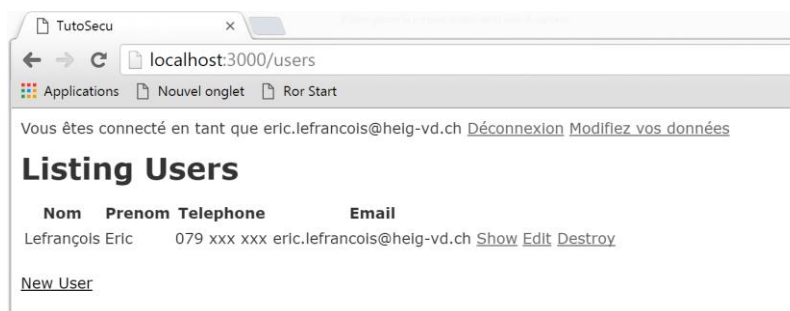
```
def user_params
  params.require(:user).permit(:nom, :prenom, :telephone, :email)
end
```

Vous devez obtenir (avec l'email que vous avez saisi) :



- Avec le lien « Edit », vous pouvez modifier vos données personnelles.
- Avec le lien « Modifiez vos données », vous pouvez modifier votre mot de passe et également votre email.

Après avoir saisi vos données personnelles, vous obtiendrez :



2 Compléments Devise

Le plugin Devise met à disposition beaucoup d'options mais aussi diverses approches d'implémentation.

Comme il n'est pas possible de toutes les lister dans ce tutorial, nous avons présenté la plus simple qui a le mérite de mettre en place un système rapidement. Toutefois, suivant les besoins de votre application, d'autres approches ou configurations seront sans doute plus adaptées.

Le cas échéant, prenez la peine de jeter un coup d'œil rapide ici:

<https://github.com/plataformatec/devise>

Voici toutefois quelques extensions intéressantes.

2.1 Choix de l'identifiant

Nous avons vu que Devise utilise de base l'adresse e-mail comme identifiant, il est cependant possible d'utiliser autre chose, comme par exemple un « username ».

Pour ce faire, il suffit d'ajouter un attribut `username` à la table `users` et le définir comme étant le login à utiliser dans le fichier de configuration de Devise (`/config/initializer/devise.rb`). Ensuite, modifier les formulaires d'inscription et d'édition, et enfin, ne pas oublier de rendre l'attribut accessible.

La marche à suivre est décrite en détails ici:

<https://github.com/plataformatec/devise/wiki/How-To:-Allow-users-to-sign-in-with-something-other-than-their-email-address>

2.2 Configurer les options de Devise

Nous pouvons modifier la quasi-totalité des options utilisées par Devise dans le fichier `/config/initializer/devise.rb`.

Par exemple lors de l'inscription, vous aurez peut-être remarqué que le mot de passe est soumis à quelque contrainte de taille ou de caractère.

Sign up

1 error prohibited this user from being saved:

- Password is too short (minimum is 8 characters)

— ..

Il est possible de régler ces paramètres dans le fichier même de configuration, ainsi que quantité de petits détails relatifs à l'authentification :

- Caractères acceptés pour le mot de passe ;
- Temps d'inactivité après lequel l'utilisateur est automatiquement déconnecté ;
- Les options à saisir (e-mail, nom, question secrète,...) afin de récupérer un mot de passe que l'on a oublié ;
- Etc..

Il est également possible de changer les messages affichés par Devise.

Les textes de ces messages sont spécifiés dans `config/locales/devise.en.yml` dont voici un extrait :

```
en:
  devise:
    confirmations:
      confirmed: "Your email address has been successfully confirmed."
      send_instructions: "You will receive an email with instructions for
how to confirm your email address in a few minutes."
```

3 Ajouter des rôles

Où en sommes-nous ? Un petit bilan..

En l'état, nous avons utilisé la librairie **Devise** pour :

- compléter le modèle `User` avec les différentes données d'authentification
- créer les formulaires utilisés pour le Sign-in, Sign-up et la modification des données d'authentification.

Notre application nous permet ainsi de gérer les utilisateurs et de les authentifier.

Notre objectif consiste maintenant à ajouter des rôles et à définir les fonctionnalités qui seront autorisées à chacun de ces rôles.

Cette partie du tutorial est indépendante de **Devise**.

3.1 Ajouter des rôles

Pour ce faire, nous allons simplement mettre en place le scaffold d'un gestionnaire de rôles, qui s'appuiera sur une table `roles` contenant une simple et unique colonne enregistrant le nom du rôle.

```
rails generate scaffold Role name:string
```

3.2 Mise en place des relations entre rôles les utilisateurs

La relation entre utilisateurs et rôles est de type n-n. Un même utilisateur pourra jouer plusieurs rôles. Nous aurons donc besoin d'une table d'association "`roles_users`" que nous allons créer au moyen d'un fichier de migration.

```
rails generate migration RolesUtilisateurs
```

Il nous reste à compléter ce fichier de migration avec le code suivant:

```
class RolesUtilisateurs < ActiveRecord::Migration
  def change
    create_table :roles_users, :id => false do |t|
      t.references :role
      t.references :user
    end
  end
end
```

Après tout ça, on peut alors exécuter les migrations en attendant `db:migrate`.

```
rake db:migrate
```

Et, enfin, définir l'association n-n dans les modèles `User` et `Role`:

```
class Role < ActiveRecord::Base
  has_and_belongs_to_many :users
end
```



Dans la classe `User`, rajouter comme illustré ci-dessous :

- o La clause `has_and_belongs_to_many`
- o Une méthode `has_role?(role)` qui permettra de contrôler si un utilisateur possède tel ou tel rôle.

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :roles

  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  def has_role?(role)
    return self.roles.find_by(:name => role.to_s.camelize)
  end
end
```



Note sur “camelize”

By default, `camelize` converts strings to `UpperCamelCase`. If the argument to `camelize` is set to `:lower` then `camelize` produces `lowerCamelCase`. `camelize` will also convert `'/'` to `':'` which is useful for converting paths to namespaces.

```
'active_model'.camelize           # => "ActiveModel"
'active_model'.camelize(:lower)    # => "activeModel"
'active_model/errors'.camelize     # => "ActiveModel::Errors"
'active_model/errors'.camelize(:lower) # => "activeModel::Errors"
```

Dans le contrôleur « `users_controller.rb` », rajouter les deux actions suivantes :

```
def edit_roles
  @user = User.find(params[:id])
end

def save_roles
  @user = User.find(params[:id])
  @user.roles.each do |le_role|
    @user.roles.delete(le_role)
  end
end
```

```

end
if params[:user]
  for id_role in params[:user][:role_ids]
    le_role = Role.find(id_role)
    @user.roles << le_role
  end
end

redirect_to users_path
end

```

Dans le fichier de routage `routes.rb`, ajouter les deux routes correspondantes au routage de ressources « users » :

```

resources :users do
  member do
    put :save_roles
    get :edit_roles
  end
end

```

Changeons la vue index de l'utilisateur (`views/User/index.html.erb`):

- Nous supprimons le lien de création d'utilisateur (l'utilisateur est créé au moment de son enregistrement)
- Un lien d'édition est rajouté, pointant sur l'action `edit_roles`

```

<h1>Liste des utilisateurs</h1>

<table>
  <tr>
    <th>Nom</th>
    <th>Prenom</th>
    <th>Email</th>
    <th>Telephone</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

  <% @users.each do |user| %>
    <tr>
      <td><%= user.nom %></td>
      <td><%= user.prenom %></td>
      <td><%= user.email %></td>
      <td><%= user.telephone %></td>
      <td><%= link_to 'Show', user %></td>
      <td><%= link_to 'Edit', edit_user_path(user) %></td>
      <td><%= link_to 'Edit Rôles', edit_roles_user_path(user) %></td>
      <td><%= link_to 'Destroy', user, confirm: 'Are you sure?', method:
:delete %></td>
    </tr>
  <% end %>
</table>

<br />

```

```
<!--<%= link_to 'Nouvel utilisateur', new_user_path %>-->
```

Rajouter la vue « `views/User/edit_roles.html.erb` ». Cette vue permet d'éditer les rôles d'un utilisateur au moyen d'une liste de boîtes à cocher. Au moment où le formulaire est soumis, l'url pointe sur l'action `save_roles` du contrôleur `UserController`.

```
<h2>Specification des rôles joués par <%= @user.prenom %> <%= @user.nom %> </h2>
<%= form_for(@user, :url => save_roles_user_path(@user), method: :put) do |f| %>

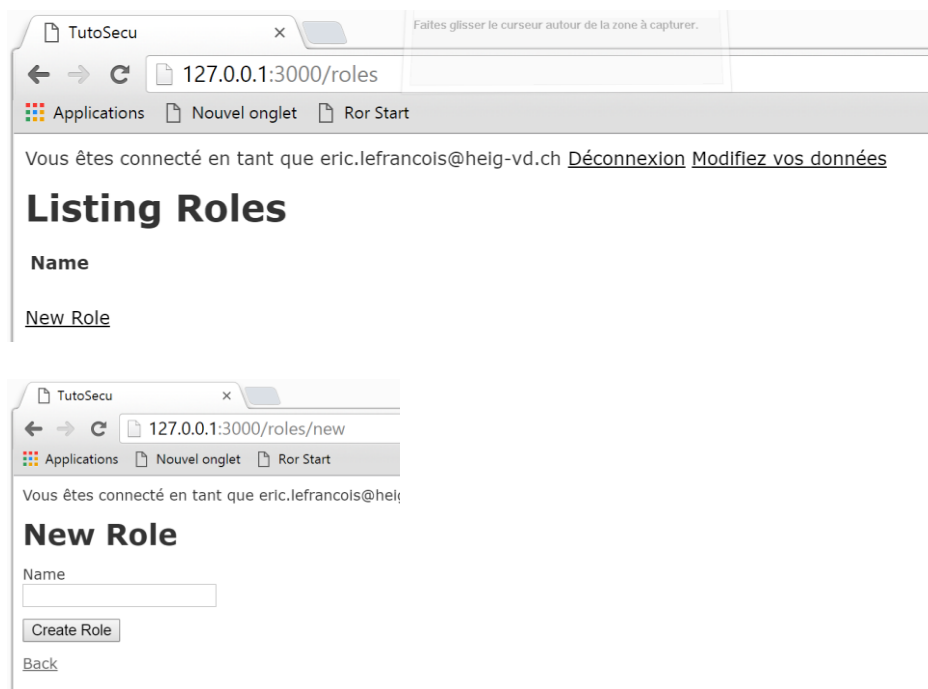
  <% for role in Role.all %>
    <div>
      <%= check_box_tag "user[role_ids][]", role.id, @user.roles.include?(role) %>
      <%= role.name %>
    </div>
  <% end %>

  <div class="actions">
    <%= f.submit "Sauvegarder" %>
  </div>

<% end %>
```



Pour tester, encore faut-il avoir défini des rôles...



Saisissez deux rôles, admin et utilisateur



La forme `user[role_ids][]` utilisée dans les champs de type boîte à cocher aura pour effet de retourner dans la requête http des query strings sous la forme `user[roles][nomrôle]`. Au moment de la sauvegarde, l'analyse des attributs par la méthode `save` aura pour effet de mettre à jour la table d'association `users_roles` en

tenant compte des paramètres reçus. save s'appuie pour ce faire sur la clause has_and_belongs_to_many.

Il nous reste également à mettre à jour les vues `index.html.erb` et `show.html.erb` de `app/views/User` de manière à afficher les rôles de chaque utilisateur. On vous confie cette tâche..



Un conseil.. Rajoutez la méthode suivante dans le modèle `User` (limitez ainsi le code dans les vues et le contrôleur !!)

```
# Retourne la liste des rôles sous forme d'un texte
def roles_as_text
  texte=''
  self.roles.each do |role|
    texte=texte + role.name + " "
  end
  return texte
end
```

A ce stade, nous pouvons donc rajouter des nouveaux utilisateurs et leur attribuer des rôles. Mais nous n'avons pas encore spécifié les permissions se rattachant à chacun des rôles. C'est là que `CanCanCan` entre en jeu.

4 CanCanCan

Les autorisations associées à chaque utilisateur sont à définir dans la classe `Ability`.

CanCanCan met à disposition un générateur pour Rails 3, qui permet de créer cette classe dans le fichier `app/models/ability.rb` (sinon, créer ce fichier manuellement) :

```
rails g cancan:ability
```

4.1 La définition des droits dans `ability.rb`

Complétez le fichier `ability.rb` pour obtenir quelque chose ressemblant à :

```
class Ability
  include CanCan::Ability

  def initialize(user)
    # Define abilities for the passed in user here. For example:
    #
    # user ||= User.new # guest user (not logged in)
    if user.has_role?(:admin)
      can :read, :all
      can :manage, User do |un_user|
        !un_user.has_role?(:admin) ||
          un_user.id==user.id
      end
      can :voir_roles, User
    end

    if user.has_role?(:utilisateur)
      can :read, :all
    end
  end
end
```

Les explications arrivent.. 😊

De manière très générale, nous définissons les droits par l'instruction:

```
can :Droit, ClasseObjetConcernée
```

Ce qui nous permettra, dans le code, d'écrire des choses qui ressembleront à :

*SI l'utilisateur courant possède le droit « **un droit** » sur l'objet « x » ALORS
Code spécifique pour les ayant droit
FINSI*

:Droit prédéfinis:

- o :read=> Ce droit regroupe les droits « index » et « show » sur la classe d'objet concernée (2^{ème} paramètre)
- o :create=> Ce droit regroupe les droits new et show sur la classe d'objet concernée (2^{ème} paramètre).
- o :update=> Ce droit regroupe les droits edit et update sur la classe d'objet concernée (2^{ème} paramètre).
- o :destroy=> Ce droit regroupe le droit destroy sur la classe d'objet concernée (2^{ème} paramètre).
- o :manage=> Raccourci d'écriture. Le droit :manage regroupe les 4 droits :create,:update,:read et:destroy

A savoir, les 7 droits RestFull index, show, new, create, edit, update, destroy sur la classe d'objet concernée (2^{ème} paramètre)

ClasseObjetConcernée prédéfinie :

- o :all=> Le droit concerné s'applique à n'importe quelle classe d'objets.

Des explications pour notre exemple..

```
if user.has_role?(:admin)
  can :read, :all
```

⇒ Un utilisateur ayant le rôle admin possède le droit de lecture (à savoir show et index) sur n'importe quel type d'objet.

```
can :manage, User do |un_user|
  !un_user.has_role?(:admin) || un_user.id==user.id
end
```

⇒ Un utilisateur ayant le rôle admin possède tous les droits RESTFull (à savoir new, create,edit, update, show, index et destroy) sur tous les objets utilisateurs pour autant qu'il ne s'agisse pas d'admin. Il peut toutefois opérer toutes ces actions si l'utilisateur admin en question n'est autre que lui-même.

Voir le paragraphe [5.2.3 Conditions supplémentaires](#) pour plus d'informations sur les conditions permettant de restreindre les enregistrements sur lesquels les permissions sont appliquées.

```
can :voir_roles, User
```

⇒ Un utilisateur ayant le rôle admin possède le droit « **voir_roles** » sur n'importe quel objet de type User.

Le droit « **voir_roles** » est un droit que l'on a « inventé ». Il n'est pas prédéfini comme le sont les droits index, show, new, create etc..

end



En bref, si on résume, on a dit dans `ability.rb`:

- qu'un rôle **admin** possède:
 - Le droit "read" sur n'importe quel type d'objet, à savoir les droits "show" et "index"
 - Tous les droits RESTFull sur lui-même (un objet `User` admin) ou sur tous les objets `User` qui ne sont pas des admins.
 - Le droit « voir_roles » sur n'importe quel objet `User`
- qu'un rôle **utilisateur** possède le droit "read" sur n'importe quel type d'objet.

Pour plus d'informations sur les différents droits (`:manage`, `:read`, `:update`,...), se référer à l'annexe du même document: [5 Annexe - Documentation CanCanCan](#), ou à cette page => <https://github.com/CanCanCommunity/cancancan>

Il ne nous reste plus qu'à aller protéger nos méthodes dans le contrôleur.

4.2 Utiliser les droits pour contrôler l'accès aux actions

Protégeons par exemple l'action `show`, afin qu'elle ne soit exécutable que par les utilisateurs ayant le rôle admin ou utilisateur :

```
# GET /users/1
# GET /users/1.json
def show
  @user = User.find(params[:id])  ⇨ Ne pas écrire cette instruction!!
  Cette dernière est implicite grace à
  l'instruction before_action :set_user, only: [:show, :edit, :update,
  :destroy] écrite au début du contrôleur

  authorize! :read, @user, :message => "Vous n'avez pas l'autorisation "

```



Contrôlez l'interdiction de l'action `show` en vous enregistrant sous un nouveau login-password et en vous enlevant tous les rôles !

Nous verrons plus loin comment contrôler, en une seule fois, l'accès à l'ensemble des actions RestFULL

4.3 Utiliser les droits dans le code des vues

Nous allons maintenant enlever des éléments que nous ne souhaitons pas afficher aux utilisateurs qui n'y ont pas accès, par exemple si nous ne souhaitons pas afficher la possibilité d'éditer les rôles ou de supprimer un utilisateur à quelqu'un qui ne le peut pas:

Fichier `views/User/index.html.erb`

```
<h1>Liste des utilisateurs</h1>

<table>
  <tr>
    <th>Nom</th>
    <th>Prenom</th>
    <th>Email</th>
    <th>Telephone</th>

    <% if can? :voir_roles, @user%>
      <th>Rôles</th>
    <% end %>

    La colonne Rôle est affichée aux seuls utilisateurs qui possèdent le droit « voir_roles » sur le
    type d'objet User

    <th></th>
    <th></th>
    <th></th>
  </tr>

  <% @users.each do |user| %>
    <tr>
      <td><%= user.nom %></td>
      <td><%= user.prenom %></td>
      <td><%= user.email %></td>
      <td><%= user.telephone %></td>
      <% if can? :voir_roles, user%>
        <td><%= user.roles_as_text %></td>
      <% end %>
      Seuls les utilisateurs qui possèdent le droit :voir_roles pour l'utilisateur en question verront
      cette colonne
      <td><%= link_to 'Show', user %></td>

      <% if can? :manage, user %>
        <td><%= link_to 'Edit Rôles', edit_roles_user_path(user) %></td>
        <td><%= link_to 'Destroy', user, confirm: 'Are you sure?',
          method: :delete %></td>
      <% end%>

      Seuls les utilisateurs qui possèdent le droit :manage pour l'utilisateur en question
      ont la possibilité de détruire cet utilisateur ou d'éditer ses rôles.

    </tr>
  <% end %>
</table>

<br />
```

```
<!--<%= link_to 'Nouvel utilisateur', new_user_path %>-->
```



Essayez.. Mais attention !! Conservez au moins un utilisateur qui possède les droits admin

Fichier `views/User/show.html.erb`

```
<p id="notice"><%= notice %></p>

<p>
  <b>Nom:</b>
  <%= @user.nom %>
</p>

<p>
  <b>Prenom:</b>
  <%= @user.prenom %>
</p>

<p>
  <b>Telephone:</b>
  <%= @user.telephone %>
</p>

<% if can? :voir_roles, User %>
<p>
  <b>Rôles:</b>
  <%= @user.roles_as_text %>
</p>
<% end %>

  Les rôles de l'utilisateur sont affichés à condition de posséder le droit
  « voir_roles » sur le type d'objet User

<%= link_to 'Back', users_path %>
```



Voici donc le résultat que vous devriez avoir côté admin, en supposant que Eric Lefrançois est connecté :

← → ↻ 127.0.0.1:3000

Applications | Nouvel onglet | Ror Start

Vous êtes connecté en tant que eric.lefrancois@heig-vd.ch [Déconnexion](#) [Modifiez vos données](#)

Signed in successfully.

Listing Users

Nom	Prenom	Telephone	Email	Roles	
Lefrançois	Eric	079 xxx xxx	eric.lefrancois@heig-vd.ch	admin	Show Edit Edit Rôles Destroy
France	Anatole		a.france@gmail.com	utilisateur	Show Edit Edit Rôles Destroy
Dupont	Paul		paul.dupont@gmail.com	admin utilisateur	Show Edit
Droit	Sans		sans.droit@gmail.com		Show Edit Edit Rôles Destroy

Eric Lefrançois, en tant qu'admin, possède le droit d'éditer les rôles de lui-même ainsi que d'Anatole France, mais pas de Paul Dupont qui est un administrateur.



Et, du côté utilisateur (Anatole France connecté) :

← → ↻ 127.0.0.1:3000

Applications Nouvel onglet Ror Start

Vous êtes connecté en tant que a.france@gmail.com [Déconnexion](#) [Modifiez vos données](#)

Signed in successfully.

Listing Users

Nom	Prenom	Telephone	Email	
Lefrançois	Eric	079 xxx xxx	eric.lefrancois@heig-vd.ch	Show Edit
France	Anatole		a.france@gmail.com	Show Edit
Dupont	Paul		paul.dupont@gmail.com	Show Edit
Droit	Sans		sans.droit@gmail.com	Show Edit



Enfin, du côté « visiteur » (Sans Droit connecté) :

Interface identique à celle de l'utilisateur. Un click sur `show` lèvera cependant une exception.

← → ↻ 127.0.0.1:3000

Applications Nouvel onglet Ror Start

Vous êtes connecté en tant que sans.droit@gmail.com [Déconnexion](#) [Modifiez vos données](#)

Signed in successfully.

Listing Users

Nom	Prenom	Telephone	Email	
Lefrançois	Eric	079 xxx xxx	eric.lefrancois@heig-vd.ch	Show Edit
France	Anatole		a.france@gmail.com	Show Edit
Dupont	Paul		paul.dupont@gmail.com	Show Edit
Droit	Sans		sans.droit@gmail.com	Show Edit

4.4 Utiliser les droits pour contrôler l'accès à l'ensemble des actions RestFULL

Pour finir, une commodité offerte par CanCanCan..



Permettant de contrôler automatiquement toutes les actions « RESTFull » du contrôleur..

La méthode `load_and_authorize_resource` est mise à disposition afin de contrôler automatiquement l'accès à toutes les méthodes REST Full d'un contrôleur REST Full (à savoir les actions : `:index`, `:show`, `:new`, `:create`, `:destroy` et `:update`).



Essayez !!

```
class UsersController < ApplicationController

  before_filter :authenticate_user!
  load_and_authorize_resource :except => :index

  # GET /users
```

```
# GET /users.json
[..]
```

Dans ce cas de figure, toutes les actions sont contrôlées à l'exception de `index`.



Ecrire l'instruction **`load_and_authorize_resource`** a le même effet que si nous avions rajouté les différentes instructions `authorize!` correspondantes dans le code comme ci-dessous :

```
def show
  authorize! :read, @user
end

def new
  @user = User.new
  authorize! :create, @user
end

def edit
  authorize! :update, @user
end

def create
  @user = User.new(user_params)
  authorize! :create, @user
  [..]
end

def update
  authorize! :update, @user
  [..]
end

def destroy
  @user.destroy
  authorize! :destroy, @user
  [..]
end
```

Ainsi, un utilisateur qui ne possède ni le rôle `admin` ni le rôle `utilisateur` (un simple visiteur) peut uniquement avoir accès à la méthode `index`, et donc voir la liste des utilisateurs.

Cette facilité ne s'adresse qu'aux actions `RestFULL`, les autres actions doivent être munies, chacune, de leur propre contrôle avec un **`authorize!`**. Voir le paragraphe 4.2 Utiliser les droits pour contrôler l'accès aux actions.


Depuis cet `index`, si un tel utilisateur sélectionne le lien « `show` », une exception sera levée de type `CanCan::AccessDenied`. En effet, l'action `show` est contrôlée..

Si une demande d'autorisation échoue, l'exception `CanCan::AccessDenied` est aussitôt levée. Il est possible de récupérer cette exception et de modifier son comportement au niveau de la classe `ApplicationController`.

```
class ApplicationController < ActionController::Base

  rescue_from CanCan::AccessDenied do |exception|
    redirect_to root_url
  end

end
```


 Essayez !!

Nous sommes arrivés au bout !

5 Annexe - Documentation CanCanCan

5.1 Installation de CanCanCan

Dans **Rails 4**, ajouter la ligne ci-dessous dans votre fichier `Gemfile`.

```
gem "cancancan"
```

Puis exécuter la commande `bundle install`.

5.2 Comment démarrer

CanCanCan s'attend à ce que la méthode `current_user` existe au niveau de chaque contrôleur. Cette méthode doit être mise à disposition par l'installation d'un module d'authentification comme Authlogic ou Devise.

■ Définir les permissions

Les autorisations associées à chaque utilisateur sont à définir dans la classe `Ability`. CanCan met à disposition un générateur pour Rails 3, qui permet de créer cette classe.

```
rails g cancan:ability
```

Voir plus loin [5.2.1 Définir les permissions](#)

■ Contrôler les permissions

Les permissions de l'utilisateur courant peuvent être contrôlées via les méthodes `can?` et `cannot?` accessibles tant au niveau du contrôleur qu'au niveau des vues.

```
<% if can? :update, @article %>
  <%= link_to "Edit", edit_article_path(@article) %>
<% end %>
```

Voir [5.4 Contrôler les permissions](#) pour plus de détails.

La méthode `authorize!`, que l'on peut utiliser dans le contrôleur, permettra de lever une exception si l'utilisateur courant n'a pas le droit d'exécuter l'action demandée.

```
def show
  @article = Article.find(params[:id])
  authorize! :read, @article
end
```

On peut écrire l’instruction `authorize!` pour chacune des actions, mais c’est un peu laborieux !

Aussi, la méthode `load_and_authorize_resource` est mise à disposition afin de contrôler automatiquement toutes les actions dans un contrôleur de type RESTful (à savoir un contrôleur mettant à disposition les méthodes usuelles `:index`, `:show`, `:new`, `:create`, `:destroy`, `:update` avec le routage correspondant).

```
class ArticlesController < ApplicationController
  load_and_authorize_resource

  def show
    # @article is already loaded and authorized
  end
end
```

Voir [5.6 Autoriser les actions du contrôleur](#) pour plus de détails

■ Gérer les accès non autorisés

Si une demande d’autorisation échoue, l’exception `CanCan::AccessDenied` est aussitôt levée. Il est possible de récupérer cette exception et de modifier son comportement au niveau de la classe `ApplicationController`.

```
class ApplicationController < ActionController::Base
  rescue_from CanCan::AccessDenied do |exception|
    redirect_to root_url, :alert => exception.message
  end
end
```

■ Généraliser le contrôle

Si on désire enfin que chaque action de l’application soit contrôlée par une demande d’autorisation, il suffit d’ajouter `check_authorization` dans l’`ApplicationController`.

```
class ApplicationController < ActionController::Base
  check_authorization
end
```

Par la suite, si on désire court-circuiter ce contrôle pour tel ou tel contrôleur, il suffit alors d’ajouter `skip_authorization_check` dans le contrôleur concerné.

5.2.1 DEFINIR LES PERMISSIONS

C’est dans la classe `Ability` que sont définies toutes les permissions. Voici un exemple.

```
class Ability
  include cancan::Ability
end
```

```

def initialize(user)
  user ||= User.new # guest user (not logged in)
  if user.admin?
    can :manage, :all
  else
    can :read, :all
  end
end
end
end

```

L'utilisateur courant est passé en paramètre de la méthode `initialize`. Ainsi, les autorisations peuvent être modifiées en se basant sur n'importe quel attribut de l'objet utilisateur. `CanCanCan` ne fait aucune supposition sur la manière dont sont gérés les rôles au niveau de l'application.

Dans cet exemple, on suppose qu'une méthode `admin?` a été mise en place afin de déterminer si l'utilisateur courant possède le rôle d'administrateur.

5.2.2 LA METHODE CAN

La méthode `can` est utilisée pour définir les autorisations. Cette méthode requiert deux arguments.

- Le premier correspond à l'action pour laquelle on désire attribuer des autorisations.
- Le deuxième dénote la classe de l'objet sur lequel on définit l'autorisation.

```
can :update, Article
```

Il est possible de passer les paramètres `:manage` pour dire que cela représente n'importe quelle action et `:all` pour représenter n'importe quel type d'objet.

```

can :manage, Article # L'utilisateur peut accomplir n'importe quelle action sur les articles
can :read, :all      # L'utilisateur peut lire n'importe quel objet
can :manage, :all    # L'utilisateur peut faire ce qu'il veut sur n'importe quel objet

```

A priori, les actions préféfinies possibles sont `:read` **`:create`** **`:update`** **`:destroy`**

- **`:read`** → `index`, `show`
- **`:create`**, → `new`, `create`
- **`:update`** → `edit`, `update`
- **`:destroy`**, → `destroy`

■ Permissions définies par le programmeur

Mais il est possible d'autres permissions: nous ne sommes pas limités aux 7 actions REST Full!

Par exemple, si l'on décide que seuls les administrateurs auront la possibilité "d'assigner les rôles", on pourrait le faire ainsi :

```

# in models/ability.rb
can :assign_roles, User if user.admin?

```

On pourra alors contrôler dans une vue que l'utilisateur a bien les permissions requises avant d'afficher les boîtes à cocher qui permettront d'assigner les rôles.

```
<!-- users/_form.html.erb -->
<% if can? :assign_roles, @user %>
  <!-- role checkboxes go here -->
<% end %>

# users_controller.rb
def update
  authorize! :assign_roles, @user if params[:user][:assign_roles]
  # ...
end
```

Maintenant, seuls les administrateurs auront la possibilité d'assigner des rôles.

■ Tableaux de permissions

Il est possible d'utiliser des tableaux. Dans l'exemple ci-dessous, l'utilisateur aura la possibilité de mettre à jour et détruire tous les objets de type `Article` et `Comment`.

```
can [:update, :destroy], [Article, Comment]
```

5.2.3 CONDITIONS SUPPLEMENTAIRES

Il est possible de spécifier un ensemble de conditions au moyen d'un hash. Ces conditions permettront de restreindre un peu plus les enregistrements sur lesquels les permissions sont appliquées.

Dans l'exemple ci-dessous, l'utilisateur aura des permissions uniquement sur les projets actifs qu'il possède.

```
can :read, Project, [:active => true, :user_id => user.id]
```

Il est alors nécessaire de n'utiliser que les colonnes de la base de données pour que ces conditions puissent être utilisées.

■ Conditions sur les associations

Il est encore possible d'utiliser des hash imbriqués pour spécifier les conditions sur des associations.

Dans l'exemple ci-dessous, le projet peut être lu que si la catégorie à laquelle il appartient est visible.

```
can :read, Project, [:category => { :visible => true }]
```

Un tableau ou un intervalle peut être spécifié en cas de valeurs autorisées multiples.

```
can :read, Project, :priority => 1..3
```

■ Cas complexes

Si vous avez des cas complexes qui ne peuvent être mis en oeuvre au moyen d'un hash de conditions, voir le paragraphe [5.3 Définir des permissions avec des blocs](#).

5.2.4 COMBINER LES PERMISSIONS

Il est possible de définir plusieurs permissions pour la même ressource. Ci-dessous, l'utilisateur va pouvoir lire des projets qui sont terminés (`released`) OU qui sont à même d'être contrôlés (`preview`).

```
can :read, Project, :released => true
can :read, Project, :preview => true
```

5.2.5 LA MÉTHODE `cannot`

La méthode `cannot` utilise les mêmes arguments que `can` pour définir les actions que l'utilisateur ne peut pas accomplir. C'est fait normalement après un appel `can` plus générique.

```
can :manage, Project cannot :destroy, Project
```

L'ordre de ces commandes est important. Voir [Ability Precedence](#) pour plus de détails.

5.3 Définir des permissions avec des blocs

Si les conditions sont trop compliquées pour être définies avec un hash, il est possible d'utiliser un bloc.

```
can :update, Project do |project|
  project.priority < 3
end
```

Si le bloc retourne `true`, l'utilisateur aura la permission, sinon pas.



Uniquement pour les attributs d'objets !

Le bloc est évalué uniquement quand l'instance de l'objet est présente. Il n'est pas évalué lorsque l'on teste des permissions au niveau de la classe, ce qui est le cas pour les actions `index` ou `new`.

Cela signifie que toute permission qui n'est pas dépendante des attributs de l'objet n'a rien à faire dans le bloc.

5.4 Contrôler les permissions

Après avoir défini les permissions, on peut utiliser la méthode `can?` dans le contrôleur pour contrôler les permissions pour une action donnée et un objet donné.

```
can? :destroy, @project
```

La méthode `cannot?` accomplit le contrôle inverse de celui de `can?`

```
cannot? :destroy, @project
```

■ Contrôler avec une classe

Il est possible également de passer la classe plutôt que l'instance (si on n'a pas d'instance à disposition).

```
<% if can? :create, Project %>
  <%= link_to "New Project", new_project_path %>
<% end %>
```



Important

Si un bloc ou un hash de conditions opère un contrôle, ce dernier retournera `true` de toute manière.

Par exemple, supposons:

```
can :read, Project, :priority => 3
```

Si on écrit:

```
can? :read, @projet # OK, le contrôle se fait sur la priorité du projet
```

Si maintenant on écrit plutôt:

```
can? :read, Project # le contrôle retournera true !!
```

En effet, il est impossible de répondre à la question `can?` Car on ne connaît pas le projet. La classe ne dispose pas d'un attribut `priority` qui permettrait de faire le check.

C'est comme si on posait la question: « Est-ce que l'utilisateur courant peut lire un projet ? ». La réponse est oui. Mais en fait ça dépend du projet (de sa priorité). On peut toujours faire un contrôle de classe, mais il faut refaire plus tard un contrôle au niveau de l'instance dès que l'instance devient disponible.

Ce comportement peut s'expliquer en raison de l'existence de l'action `index`.

Puisque que le “before filtre” `authorize_resource` n'a pas d'instance à contrôler, le check est opéré au niveau de la classe. Si le check retournerait `false`, il ne serait pas possible, plus tard de faire des checks spécifiques au niveau des instances.

C'est la raison pour laquelle le fait de passer une classe à `can?` retourne `true`.

5.5 Tester les permissions autrement (Functional testing)

Dans certains cas, il peut s'avérer difficile de tester les permissions des utilisateurs quand on se trouve à un niveau fonctionnel ou d'intégration car on n'a pas forcément l'utilisateur courant sous la main..

Toutefois, comme **CanCan** gère toutes les permissions au moyen de la classe `Ability`, il devient facile par exemple de mettre en place tout un ensemble complet de tests d'unités.

La méthode `can?` Peut être invoquée directement sur une instance de la classe `Ability` (comme on le ferait normalement depuis un contrôleur ou une vue). Ainsi, il est très facile de contrôler la logique de permissions.

```
test "user can only destroy projects which he owns" do
  user = User.create!
  ability = Ability.new(user)
  assert ability.can?(:destroy, Project.new(:user => user))
  assert ability.cannot?(:destroy, Project.new)
end
```

5.6 Autoriser les actions du contrôleur

On peut utiliser la méthode `authorize!` pour contrôler manuellement les autorisations au niveau du contrôleur. L'exception `CanCan::AccessDenied` sera levée dans les cas où l'utilisateur n'a pas les permissions requises. Voir [5.7 Exception handling](#) pour connaître les moyens de réagir à ce genre d'événement.

```
def show
  @project = Project.find(params[:project])
  authorize! :show, @project
end
```

Aussi, la méthode `load_and_authorize_resource` est mise à disposition afin de contrôler les méthodes automatiquement toutes les actions dans un contrôleur de type RESTful (à savoir un contrôleur mettant à disposition usuelles `:index`, `:show`, `:new`, `:create`, `:destroy`, `:update` avec le routage correspondant).

Notons que cette méthode utilise un “before filter” pour charger la ressource dans une variable d'instance et puis invoque la méthode `authorize!` pour cette ressource.

```
class ProductsController < ActionController::Base
  load_and_authorize_resource
end
```

Ce qui revient au même que d'écrire:

```
class ProductsController < ActionController::Base
  load_resource
  authorize_resource
end
```

Depuis **CanCan** 1.5, on peut utiliser `skip_load_and_authorize_resource`, `skip_load_resource` or `skip_authorize_resource` permet de court-circuiter et de spécifier de actions spécifiques comme dans un before filter. Par exemple.

```
class ProductsController < ActionController::Base
  load_and_authorize_resource
  skip_authorize_resource :only => :new
end
```


Voir aussi [Controller Authorization Example](#), [Ensure Authorization](#) et [Non RESTful Controllers](#).

■ Choisir les Actions

Par défaut, cela va s'appliquer à toutes les actions du contrôleur, même celles qui ne font pas partie des 7 actions RESTful. Le nom de l'action sera passée en paramètre au moment de l'autorisation. Par exemple, si on a une action `discontinue` dans `ProductsController`, cela aura ce comportement.

```
class ProductsController < ActionController::Base
  load_and_authorize_resource
  def discontinue
    # Automatically does the following:
    # @product = Product.find(params[:id])
    # authorize! :discontinue, @product
  end
end
```

On peut spécifier quelles actions sont concernées en utilisant les options `:except` et `:only`, tout comme avec les `before_filter`.

```
load_and_authorize_resource :only => [:index, :show]
```

■ Action index

La collection des ressources sera chargée entièrement en utilisant `accessible_by`.

```
def index
  # @products automatically set to Product.accessible_by(current_ability)
end
```

■ Actions show, edit, update et destroy

Ces actions rechercheront directement le bon enregistrement.

```
def show
  # @product automatically set to Product.find(params[:id])
end
```

■ Actions new et create

Ces actions initialiseront la nouvelle ressource avec les attributs trouvés dans le hash des conditions. Par exemple, si on a cette définition `can`:

```
can :manage, Product, :discontinued => false
```

Le produit qui sera construit dans le contrôleur aura cette valeur d'attribut:

```
@product = Product.new(:discontinued => false)
```

Ainsi, l'autorisation sera acceptée au moment où l'utilisateur tentera d'accéder à l'action `new`.

Les attributs sont alors ré-écrits avec les nouvelles valeurs qui seront retournées dans la variable `params (params[:product])`.

■ Nom de classe différent

Si la classe du modèle possède un espace de noms ou si encore elle est différente du nom du contrôleur, il deviendra alors nécessaire de spécifier l'option `:class`.

```
class ProductsController < ApplicationController
  load_and_authorize_resource :class => "Store::Product"
end
```

■ Recherche spécifique

Si vous désirez rechercher une ressource autrement que par l'id, il est possible de le faire au travers de l'option `find_by`.

```
load_resource :find_by => :permalink # will use find_by_permalink(params[:id])
authorize_resource
```

■ Override loading

La ressource ne sera chargée que si elle n'a pas été déjà chargée dans une variable d'instance. Cela nous permet de surcharger ce comportement au moyen d'un `before_filter` spécifique.

```
class BooksController < ApplicationController
  before_filter :find_published_book, :only => :show
  load_and_authorize_resource

  private

  def find_published_book
    @book = Book.released.find(params[:id])
  end
end
```

Il est alors nécessaire d'opérer ce chargement **avant** l'invocation de `load_and_authorize_resource`.

Si vous avez placé `authorize_resource` dans `ApplicationController`, il faut alors utiliser `prepend_before_filter` pour opérer le chargement dans les sous-classes du contrôleur de manière à ce que cela soit opéré avant la demande d'autorisation.

■ authorize_resource

Ajouter `authorize_resource` va fabriquer un "before filter" qui invoquera `authorize!`, en lui passant en paramètre la variable d'instance de la ressource si elle existe. Si cette dernière n'existe pas (comme par exemple avec l'action `index`), c'est le nom de la classe qui sera alors communiqué en paramètre. Par exemple, si on a un `ProductsController` ça le fera avant chaque action.

```
authorize!(params[:action], @product || Product)
```

5.7 Exception handling

L'exception `CanCan::AccessDenied` est levée en invoquant `authorize!` au niveau du contrôleur si l'utilisateur n'a pas les droits requis pour exécuter l'action. Il est possible de communiquer un message.

```
authorize! :read, Article, :message => "Unable to read this article."
```

L'exception peut être levée “manuellement” si on désire un comportement particulier.

```
raise CanCan::AccessDenied.new("Not authorized!", :read, Article)
```

Le message peut être personnalisé au moyen d'une internationalisation.

```
# in config/locales/en.yml
en:
  unauthorized:
    manage:
      all: "Not authorized to %{action} %{subject}."
      user: "Not allowed to manage other user accounts."
    update:
      project: "Not allowed to update this project."
```

Notez que `manage` et `all` peuvent être utilisés pour. Aussi, `%{action}` et `%{subject}` peuvent être utilisés comme variables dans le message `can be used as variables in the message`.

Il est possible de récupérer l'exception et de modifier son comportement au niveau de `ApplicationController`.

Par exemple, ci-dessous, le message est opéré par un flash et une redirection est opérée sur la page principale du site.

```
class ApplicationController < ActionController::Base
  rescue_from CanCan::AccessDenied do |exception|
    redirect_to root_url, :alert => exception.message
  end
end
```

L'action et le sujet (ressource) peuvent être retrouvés dans le code même de l'exception pour opérer un traitement ultérieur.

```
exception.action # => :read exception.subject # => Article
```

Le message d'erreur par défaut peut également être personnalisé au sein même de l'exception. Ceci dans le cas où aucun message n'est fourni en paramètre.

```
exception.default_message = "Default error message" exception.message # =>
"Default error message"
```

Si vous préférez retourner le code 403 Forbidden HTTP code, créez le fichier `public/403.html` et écrivez à une instruction `rescue_from` comme dans cet exemple dans `ApplicationController`:

```
class ApplicationController < ActionController::Base
  rescue_from CanCan::AccessDenied do |exception|
    render :file => "#{Rails.root}/public/403.html", :status => 403
  end
end
```

end

403.html doit être du pure code HTML, CSS, et JavaScript—non pas un template. Les champs de l'exception ne lui sont en effet aucunement accessibles.

5.8 Changer les valeurs par défaut

CanCan fait deux suppositions au sujet de l'application.

- Que l'on a à disposition une classe `Ability` qui définit les permissions.
- Que l'on a une méthode `current_user` dans le contrôleur qui retourne l'utilisateur courant.

Il est possible de le redéfinir en définissant la méthode `current_ability` dans `ApplicationController`. La méthode actuelle ressemble à ceci:

```
def current_ability
  @current_ability ||= Ability.new(current_user)
end
```

Mais on peut le changer !

```
# in ApplicationController
```

```
def current_ability
  @current_ability ||= AccountAbility.new(current_account)
end
```