Table des matières

1 Quoi - Où

1.1 Comment?

Le pattern **MVC** définit *quel* code il faut mettre *où*. Avec peu d'expérience il est facile de dériver et mettre du code au mauvais endroit.

Ce chapitre définit quelques règles qui permettent d'éviter cette dérive.

1.2 Fat Models, Skinny Controllers

Quel est le rôle du *controlleur* ?

C'est le code qui réceptionne les actions de l'utilisateur, prépare les données et choisi la vue à redonner à l'utilisateur.

La dérive possible ici est **prépare les données**.

En effet, cette préparation peut utiliser des algorithmes complexes, le code concerné doit se trouver au maximum dans les modèles, le moins possible dans le controlleur. D'où l'adage: **Fat models, Skinny controllers**

event.rb

```ruby
class Event < ActiveRecord::Base
  def self.between(from, to)
    # Fat code
  end

  def self.concerning(user)
    # Fat code
  end
end
```

events_controller.rb

```ruby
class EventsController < ApplicationController
  def index
    # Skinny code here!
    from = Time.now
    @events = Event.between(from, from+1.month).concerning(current_user)
  end
end
```

1.3 Dumb Views

Les vues doivent contenir uniquement du code *bête*, il n'y aura donc pas d'algorithme dedans.

Néanmoins, il arrive que pour afficher des données, il faille les mâcher ce qui n'est pas forcément le rôle du *contrôleur* ou des *modèles*.

Dans ce cas, on utilise des **helpers**. Ce sont des composants de code qui cachent la complexité à la vue.

index.html.erb

```erb
<ul>
  <% @events.each do |event| >
    <li>
      <%= color_bullet(event)><%= event %>
    </li>
  <% end >
</ul>
```

event_helpers.rb

```ruby
module EventHelper
```

```
2     EVENT_CLASSES = {'booking' => 'orange', 'appointment' => 'blue', 'party' =>
          'red'}
3
4     def color_bullet(event)
5       "<span class='event-bullet #{EVENT_CLASSES[event.kind]}'></span>"
6     end
7   end
```

2 RESTfulness

2.1 Pourquoi?

MVC a permis d'organiser son code de manière clair. Ceci permet de reprendre facilement le code d'autres développeurs.

Etant donné que la plupart des opérations que l'on fait sur ses modèles sont du CRUD, on peut aller encore plus loin en définissants nos controlleurs comme controlleur de **ressources**.

Ceci permet encore un meilleur découpage du système et donc facilite la maintenance ainsi que l'accessibilités.

2.2 C'est quoi REST?

How I explained REST to my wife

Comment j'ai expliqué REST à ma femme

2.3 URI, URL, URN, Urghhh!

Uniform **R**esource **Identifier**
Uniform **R**esource **Locator**
Uniform **R**esource **Name**

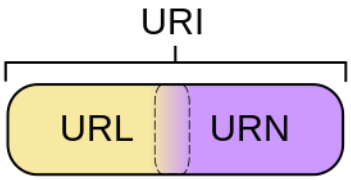

Figure 1: URI

```
                    hierarchical part
        /-------------------+--------------------\
                    authority               path
        /---------------+---------------\/---+----\
  abc://username:password@example.com:123/path/data?key=value#fragid1
  \+/   \-------+-------/ \----+----/ \+/           \---+---/ \--+--/
scheme  user information     host     port            query   fragment

  urn:example:mammal:monotreme:echidna
  \+/ \--------------+---------------/
scheme               path
```

2.4 RESTful routes

Verb	Path	Rails	Laravel	Description
GET	/resources	index	index	Display a list of resources
GET	/resources/{create,new}	new	create	Form for resource creation
POST	/resources	create	store	Create a new resource
GET	/resources/:id	show	show	Display a specific resource
GET	/resources/:id/edit	edit	edit	Form for resource edition
PUT/PATCH	/resources/:id	update	update	Update a specific resource
DELETE	/resources/:id	destroy	destroy	Delete a specific resource

2.4.1 La 8ème roue du carrosse

Chaque action modifiante possède *deux* routes:

1. Une pour présenter la resource à modifier à l'utilisateur
2. L'autre pour faire la modification proprement dite

```
GET /resource/{id}/edit
PUT /resource/{id}
```

Pour la suppression d'une resource, il n'existe que l'opération de modification:

```
DELETE /resource/{id}
```

Il nous manque donc

```
GET /resource/{id}/delete
```

Pourquoi?

_ Tout simplement parce que la plupart du temps, le lien qui pointe vers l'action de suppression demande une confirmation en javascript. Du coup, inutile de présenter la resource à supprimer avant.

Ceci nous amène donc dans certains cas à ajouter cette route manquante lorsque l'on sait que le javascript peut être non actif.

2.4.2 Ressources uniques

Certaines ressources sont uniques, les routes sont différentes puisqu'il n'y a pas d'**id**.

Verb	Path	Rails	Description
GET	/resource	show	Display the resource
GET	/resource/{create,new}	new	Form for the resource creation
POST	/resource	create	Create the resource
GET	/resource	show	Display the resource
GET	/resource/edit	edit	Form for the resource edition
PUT/PATCH	/resource	update	Update the resource
DELETE	/resource	destroy	Delete the resource

_ Cette spécificité n'est gérée nativement que par certains frameworks. Pour les autres, à vous de mettre en place une convention pour les routes et actions.

2.4.3 Ressources imbriquées (Nested resources)

Il arrive souvent qu'une ressource soit dépendante d'une autre, ou plutôt que cette ressource soit imbriquée dans une autre, deux exemples:

- `Article`: article du blog
  - `Comment`: les commentaires *liés* à cet article
- `Category`: Une catégorie de produit
  - `Product`: les produits dans cette catégorie

Fully nested routes

Verb	Path	Action	Description
GET	/articles/123/comments	index	Comments list for this article
POST	/articles/123/comments	create	Create a comment for this article
GET	/articles/123/comments/76	show	Display the single comment
GET	/articles/123/comments/76/edit	edit	Form to edit the comment

Shallow routes

Verb	Path	Action	Description
GET	/categories/23/products	index	List products of this category
POST	/categories/23/products	create	Create a product in this category
GET	/products/42	show	Display the product
GET	/products/42/edit	edit	Form to edit the product

2.5 Exemple

Si on s'en tient à des ressources *RESTful*, il est rare de devoir ajouter des `actions` personnalisées. Il suffit de trouver la **nouvelle** ressource exposée et de l'exprimer en termes **CRUD**iens.

L'exemple typique est le *login logout*, sans trop réflechir on se dit qu'il faut simplement ajouter ces deux actions dans le `UsersController`.

FAUX! Enfin pas *RESTful!*

Cette fois ci en reflechissant, le login et le logout n'ont rien à voir avec la gestion des utilisateurs (CRUD), donc pourquoi ajouter des actions dans ce controlleur?

2.5.1 Login, logout

En réalité un `login` en terme de ressource est la *création* d'une nouvelle session,
le `logout` est la *destruction* de cette session.

On se dirige donc vers un `SessionsController` avec:

Action	Description
new	Formulaire de login
create	Soumission du formulaire de login
destroy	Logout

La mise à jour d'une session n'a pas de sens, il n'a donc pas ces actions, par contre on peut imaginer ce que peut montrer les actions suivantes:

Action	Description
index	Lister les sessions courantes (les utilisateurs actuellement connectés)
show	Afficher les détails de la session: temps de connexion, type de connexion (credentials, OAuth, cookie, …)

3 API

3.1 Pourquoi?

Il est de plus en plus fréquent de pouvoir utiliser une application web depuis un autre service, donc une communication machine à machine.

En utilisant les principes REST, ceci est facile à mettre en place.

3.2 Application

En utilisant des controlleurs de ressources, nous avons les actions pour le CRUD, la seule différence est que nous devons absorber et générer du JSON ou de l'XML au lieu de l'HTML.

4 Background jobs

5 Best Practices

5.1 May the *verb* be with you

Les actions executées en réponses aux requêtes doivent respecter l'intention du verbe HTTP:

- `GET`: **Aucune modification** des données persistées ou état
- `POST`, `PUT`: Ok pour modifier des données ou états

A ne pas confondre avec les requêtes **Idempotante** (peuvent être jouées plusieurs fois sans que le résultat ne change.)

- `GET`
- `DELETE`
- parfois `PUT`

5.1.1 Exemple

Mauvais:

```
1 Route::get('/visits/{id}/delete', 'VisitsController@delete');
```

Juste:

```
1 Route::post('/visits/{id}/delete', 'VisitsController@delete');
```

5.2 Use the *route* Luke

La gestion des routes est faite par le framework, par gestion on entend:

- le dispatch
- **la génération pour les URL**

5.2.1 Exemple

Mauvais:

```
1 <form id='textedit' method="post" action="/visits/34/edit">
```

Juste:

```
1 <form id='textedit' method="post" action="<?= route('visits.edit', ['id' =>
      34]) ?>">
```

5.3 Don't underestimate the *Controller*

Les contrôleurs:

- reçoivent les requêtes
- préparent les données
- les renvoient (vue ou data)

Il n'y donc pas *forcément* une correspondance 1-1 entre modèle et contrôleur.

Même si elles gèrent les même données, il faut différencier:

- la partie *front* pour les utilisateurs ciblés
- la partie *back* (back-office) pour les gestionnaires

Nous aurons donc jusqu'à 3 contrôleurs pour une même ressource:

- `ArticlesController` (front)
- `Admin::ArticlesController` (back-office)
- `Api::ArticlesController` (API)