Créer Votre Framework Web

en RUBY

Xavier Nayrac

Créer votre framework web en Ruby

Xavier Nayrac

Ce livre est en vente à http://leanpub.com/creervotreframeworkwebenruby

Version publiée le 2020-09-14

Ce livre est publié par Leanpub. Leanpub permet aux auteurs et aux éditeurs de bénéficier du Lean Publishing. Lean Publishing consiste à publier à l'aide d'outils très simples de nombreuses itérations d'un livre électronique en cours de rédaction, d'obtenir des retours et commentaires des lecteurs afin d'améliorer le livre.

Tweet ce livre !

S'il vous plaît aidez Xavier Nayrac en parlant de ce livre sur Twitter !

Le hashtag suggéré pour ce livre est #CreerVotreFrameworkWebEnRuby.

Découvrez ce que les gens disent à propos du livre en cliquant sur ce lien pour rechercher ce hashtag sur Twitter :

#CreerVotreFrameworkWebEnRuby

Table des matières

À qui s'adresse ce livre

Ce livre s'adresse avant tout aux personnes curieuses de comprendre le fonctionnement d'un framework web et qui pour cela sont prêtes à soulever le capot et à mettre les mains dans le cambouis.

Pour tirer pleinement profit de ce livre, vous aurez besoin de quelques connaissances préliminaires en programmation. Si vous avez écrit deux ou trois sites web, quelque soit le langage, quelque soit le framework, vous possédez alors les compétences requises pour utiliser ce livre au mieux. Comme le titre le précise, nous allons écrire un framework web **en Ruby**, il vous sera donc utile d'avoir déjà une certaine connaissance du langage Ruby. Si ça n'est pas le cas, rassurez vous, Ruby est un langage qui s'apprend facilement et rapidement et vous pourrez très bien l'apprendre *en même temps que* que vous lirez ce livre.

Je vous souhaite une bonne programmation.

Introduction

Ce livre est articulé autour de 3 parties distinctes.

Dans la première partie « *Les éléments d'un framework web* » nous construirons un framework web minimal depuis zéro. Nous utiliserons, fabriquerons et manipulerons les composants essentiels à un framework web. Quelque soit le framework web que vous déciderez d'écrire, vous aurez certainement besoin de chaque élément vu dans cette première partie.

Dans la deuxième partie « *Concepts Avancés* » nous aborderons des éléments et des composants plus complexes à mettre en œuvre, qui ne sont pas essentiels à tous les frameworks en général, mais qui pourront s'avérer indispensables pour **votre** framework en particulier.

Enfin nous conclurons ce livre dans la troisième partie, avec un exemple d'application écrit de A à Z avec le framework que nous aurons mis au point Cette application sera scindée en deux : un backend et un frontend, et utilisera une API publique pour collecter et présenter des données (des GIFs ou bien des informations météo, cela reste encore à déterminer).

2 - Une route + un contrôleur + une vue = une application

Une route

Le premier composant du framework dont je voudrais discuter est le routage, *routing* en anglais.

Le rôle du routage sera d'aiguiller les requêtes venant de l'extérieur vers la bonne méthode de traitement, contenue quelque part dans l'application.

Toutes les routes de nos applications seront contenues dans le fichier `routes.yml`.

Lexique français/anglais des termes de routage

Français	Anglais
route	route
routage	routing
routeur	router

Attention à ne pas confondre *route*, la route, et *root*, la racine.

Il y a de nombreuses manières de spécifier une route, ainsi que de nombreuses techniques possibles pour charger les routes en mémoire. Dans ce livre, comme dans mon travail au quotidien, dès que je le peux je fais le choix de la simplicité. C'est pourquoi j'ai préféré utiliser un fichier de configuration au format YAML, plutôt que de définir un DSL qui aurait eu pour conséquence de trop compliquer les choses dès le début de ce livre. Mais nous reviendrons sur un DSL dans les derniers chapitres.

Voici le contenu de notre premier fichier `routes.yml`, avec la définition d'une seule route :

Votre première route

```
# routes.yml

"/hello": { via: "get", to: "hello#index" }
```

Charger le contenu d'un fichier au format YAML demande peu de travail :

Charger le fichier routes.yml avec Ruby

```
require 'yaml'
YAML.load_file("routes.yml")
#=> {
#=>     "/hello" => {
#=>         "via" => "get",
#=>          "to" => "hello#index"
#=>     }
#=> }
```

Nous récupérons donc un Hash, dont la clé principale est le chemin de la requête. La valeur est elle-même un Hash, avec une première clé `"via"` qui contient le verbe HTTP de la requête, et une seconde clé `"to"` qui contient sous une forme un peu cryptique la classe et la méthode qui effectuera le traitement voulu. Nous verrons plus loin la signification réelle de ce `"hello#index"`.

En français une telle route signifie « route le chemin `/hello`, qui vient du verbe http `GET`, vers la méthode `index` ».

Notre application va charger toutes les routes lors de son initialisation (même si pour l'instant il n'y en a qu'une) :

En route pour une nouvelle application

```
1 # application.rb
2 require 'yaml'
3
4 class Application
5
6   def initialize
7     @routes = YAML.load_file("routes.yml")
8   end
9 end
```

Lançons notre application depuis une session `irb` pour nous assurer qu'elle fonctionne. Notez que l'objet `app` qui s'affiche automatiquement dans la console contient une représentation compréhensible du Hash `@routes` :

Voir l'objet au sein d'une session irb

```
require "./application"
#=> true
Application.new
#=> #<Application:0x00559939fa2a30 @routes={"/hello"=>{"via"=>"get",
#=>                                                   "to"=>"hello#index"}}>
```

Un petit truc pour débugger

> I am a puts debuggerer[1]
> – Aaron Patterson

Si au cours d'une séance de débuggage ou pendant la mise au point interactive d'un code vous vous trouvez, dans la console, avec le besoin d'inspecter le contenu d'une variable de classe, la méthode qu'il vous faut se nomme `instance_variable_get` :

Afficher un membre

```
require "./application"
app = Application.new
app.instance_variable_get("@routes")
#=> {"/hello" => {"via" => "get", "to" => "hello#index"}}
```

J'ai rabâché plusieurs fois dans le chapitre précédent qu'une application Rack devait avoir une méthode `call`. Notre classe `Application` se doit donc de répondre à `call`. Profitons en pour tester la reconnaissance des routes. Si nous envoyons une requête avec un chemin correspondant à une route nous répondons avec « Ce chemin existe ». Si au contraire la requête ne correspond à aucun chemin nous afficherons « Ce chemin n'existe pas ».

On ajoute le call qui va bien

```
1  # application.rb
2  require 'yaml'
3
4  class Application
5
6    def initialize
7      @routes = YAML.load_file("routes.yml")
8    end
9
10   def call(env)
```

1. https://tenderlovemaking.com/2016/02/05/i-am-a-puts-debuggerer.html

```
11      if @routes[env["REQUEST_PATH"]]
12        [200, {}, ["Ce chemin existe"]]
13      else
14        [200, {}, ["Ce chemin n'existe pas"]]
15      end
16    end
17  end
```

Pour connaître la requête de l'utilisateur j'ai utilisé la variable `REQUEST_PATH`. Pourquoi celle ci et pas une autre ? Si vous avez regardé l'`env` de plus près, vous avez vu que `REQUEST_URI` semble aussi un bon candidat, au même titre que `PATH_INFO`. Sachez que j'ai choisi `REQUEST_PATH` totalement au hasard ! L'objectif actuel est que cela fonctionne avec la configuration présente, il sera temps plus tard de se pencher sur les spécifications de Rack.

Notre application comporte désormais trois fichiers.

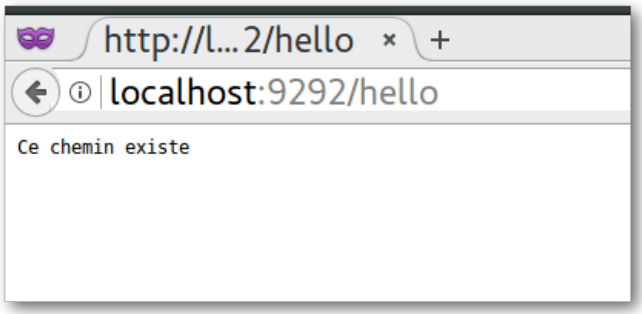
```
$ tree code/ch02/route/
├── application.rb
├── config.ru
└── routes.yml
```

Le `config.ru` est réduit à sa plus simple expression. Il se contente de charger le fichier `application.rb` et de lancer l'application Rack :

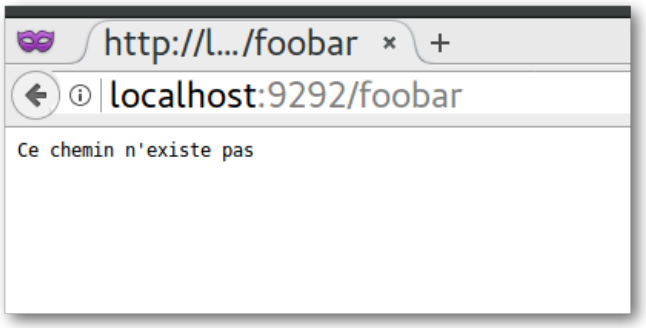
En avant !

```
1  # config.ru
2
3  require_relative "application"
4  run Application.new
```

Lancez l'application avec `rackup` et testez plusieurs routes.



Cette route existe



Celle ci n'existe pas

Un contrôleur

Nous venons de voir qu'une route devait viser une méthode précise dans notre framework. Plus particulièrement la route définit ci-dessus vise quelque chose que nous avons écrit ainsi : `"hello#index"`. La partie avant le `#` est le nom d'un contrôleur, tandis que la partie après le `#` est le nom d'une méthode de ce contrôleur.

objet#méthode

Le caractère `#` n'est pas choisi au hasard. Bien que techniquement on pourrait choisir n'importe quel caractère, le `#` en Ruby, et d'autres langages, symbolise l'appartenance d'une méthode d'instance. Au contraire du `.` qui représente l'appartenance d'une méthode de classe.

Donc `Vincent.vega` représente la méthode de classe `vega`, de la classe `Vincent`. On l'utilisera comme suit :

Une méthode de classe

```
Vincent.vega
```

Alors que `vincent#vega` représente la méthode d'instance `vega`, d'une instance `vincent`. On l'utilisera ainsi :

Une méthode d'instance

```
vincent = Vincent.new
vincent.vega
```

Le rôle d'un contrôleur sera de fabriquer la réponse attendue par Rack, à savoir un Array avec code de retour, entêtes et corps.

Sans plus attendre, voici un contrôleur minimal pour notre route :

Un contrôleur minimal, mais alors vraiment minimal

```ruby
# hello_controller.rb

class HelloController
  def index
    [200, {}, ["Coucou !"]]
  end
end
```

Vous pouvez constater que le code de ce contrôleur est terriblement simple. Nous avons quand même une convention à l'œuvre ici : *un contrôleur "foo" est définit dans une classe* `FooController`. Dans ce contrôleur nous retrouvons bien la méthode `index` qui est visée par la route `"hello#index"`. Et le code de cette méthode `index` doit vous sembler familier après avoir lu le chapitre précédent sur Rack.

Nous modifions notre application pour y inclure ce contrôleur. Nous chargeons la classe `HelloController` avec la ligne `require_relative 'hello_controller'` et nous modifions le contenu de la méthode `call` pour qu'elle utilise le contrôleur si une bonne route existe. Si le chemin de la requête ne correspond à aucune route nous exécutons `fail` et rendons le message d'erreur un peu plus explicite en affichant « No matching route » :

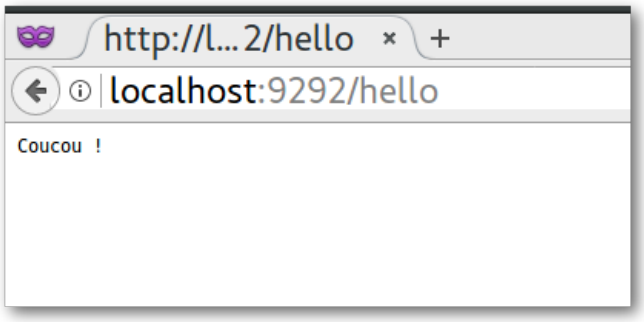
Utilisons ce fameux contrôleur

```ruby
# application.rb
require 'yaml'
require_relative 'hello_controller'

class Application

  def initialize
    @routes = YAML.load_file("routes.yml")
  end

  def call(env)
    if route_exists?(env["REQUEST_PATH"])
      HelloController.new.index
    else
      fail "Pas de route correspondante"
    end
  end

  private

  def route_exists?(path)
```

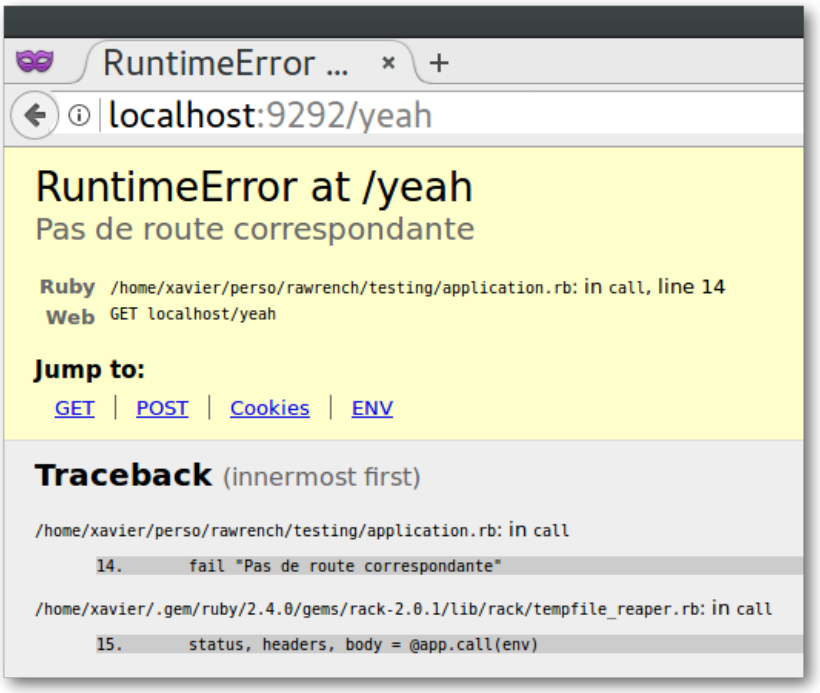
```
22      @routes[path]
23    end
24  end
```

Nous voici désormais avec un total de quatre fichiers.

```
$ tree code/ch02/controller/
├── application.rb
├── config.ru
├── hello_controller.rb
└── routes.yml
```



Quand la route existe



Quand la route n'existe pas

Une vue

Le prochain composant que nous allons introduire sera la vue. C'est ce qui nous intéresse généralement en tant qu'utilisateur du Web.

Pour l'instant nous définirons une vue comme étant ce qui s'affiche dans le navigateur.

Voici un contenu simplifié pour la vue `hello.html`. Ce n'est pas un vrai fichier HTML, complet et tout et tout, mais les navigateurs seront quand même content de l'afficher :

Soyons grandiloquent !

```
1 <!-- hello.html -->
2 <h1>Un grand bonjour depuis mon extraordinaire application</h1>
```

Nous devons transformer une vue en une chaîne de caractère, la méthode Ruby `File.read` est parfaite pour cela :

Lire la vue avec File.read

```
# hello_controller.rb
class HelloController
  def index
    [200, {}, [File.read("hello.html")]]
  end
end
```

Avec ce nouveau fichier `hello.html` nous comptons maintenant cinq fichiers.

```
$ tree code/ch02/vue/
├── application.rb
├── config.ru
├── hello_controller.rb
├── hello.html
└── routes.yml
```

Et voilà !

C'est gros, c'est beau

Faire la conversion d'une vue en une chaîne de caractère directement dans la méthode `index` du contrôleur est ce qu'il y a de plus simple. C'est pourquoi j'ai présenté les choses de cette manière jusqu'ici. Mais ça n'est pas forcement le plus intelligent à faire. Nous avons définit le rôle du contrôleur comme étant celui de façonner la réponse attendue par Rack. Mais nous n'avons pas dit que tout devait obligatoirement se dérouler *dans* le contrôleur. Nous n'avons pas interdit non plus au contrôleur de se faire aider. Isoler les responsabilités rendra notre code plus facile à tester, à modifier et à raisonner. Voici donc un code plus étoffé, faisant appel à une classe `Renderer`.

Utilisons une classe pour faire le rendu

```
# hello_controller.rb
class HelloController
  def index
    status, body = Renderer.new("hello.html").render
    [status, {}, [body]]
  end
end

# renderer.rb
class Renderer
  def initialize(filename)
    @filename = filename
  end

  def render
    if File.exists?(@filename)
      [200, File.read(@filename)]
    else
      [500, "<h1>500</h1><p>No such template: #{@filename}</p>"]
    end
  end
end
```

Le rendu d'un fichier est ainsi testable de manière isolée, sans avoir à charger ou utiliser un contrôleur dont le nombre de fonctionnalité ne manquera pas de croître. Testons ce *renderer* dans une session `irb` :

Testons la classe Renderer dans irb

```
require "./renderer"
Renderer.new("hello.html").render
#=> [
#=>   200,
#=>   "<h1>Hello from an amazing web application!</h1>\n"
#=> ]
Renderer.new("unknown").render
#=> [
#=>   500,
#=>   "<h1>500</h1><p>No such template: unknown</p>"
#=> ]
```

Erreur 500

Nous voyons ici un nouveau code de retour : 500. Ce code spécifie une erreur interne du serveur (*internal server error* en anglais). Nous devons retourner une erreur 500 lorsque le problème vient du serveur, comme par exemple une division par zéro dans notre programme, ou bien comme ici un fichier inexistant.

Vous aurez remarqué que le code de `HelloController#index` s'est un peu compliqué. Nous verrons comment le rendre beaucoup plus digeste dans le prochain chapitre.

Une application

Nous voici à la fin de ce chapitre avec une application étalée sur six fichiers. Le code est suffisamment court pour que je puisse me permettre de les reproduire ici en entier :

Le code Ruby de l'application

```ruby
# application.rb
require 'yaml'
require_relative 'hello_controller'
require_relative 'renderer'

class Application

  def initialize
    @routes = YAML.load_file("routes.yml")
  end

  def call(env)
    if route_exists?(env["REQUEST_PATH"])
      HelloController.new.index
    else
      fail "No matching routes"
    end
  end

  private

  def route_exists?(path)
    @routes[path]
  end
end

```

```ruby
# config.ru
require_relative 'application'
run Application.new


# hello_controller.rb
class HelloController
  def index
    status, body = Renderer.new("hello.html").render
    [status, {}, [body]]
  end
end


# renderer.rb
class Renderer
  def initialize(filename)
    @filename = filename
  end

  def render
    if File.exists?(@filename)
      [200, File.read(@filename)]
    else
      [500, "<h1>500</h1><p>No such template: #{@filename}</p>"]
    end
  end
end
```

Le code HTML de l'application

```html
<!-- hello.html -->
<h1>Un grand bonjour depuis mon extraordinaire application</h1>
```

Le code YAML de l'application

```yaml
# routes.yml
"/hello": { via: "get", to: "hello#index" }
```

Nous ne pouvons pas encore parler de framework. Il s'agit d'un programme. En parlant de ça, il serait bon de définir ce qu'est un *framework*. La première phrase sur Wikipédia France est une bonne introduction. Je vous invite à lire l'article entier[2], ainsi que son homologue en anglais[3].

2. https://fr.wikipedia.org/wiki/Framework
3. https://en.wikipedia.org/wiki/Software_framework

> En programmation informatique, un framework ou structure logicielle est un ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel (architecture).
> – *Wikipédia*

Une autre définition que j'aime bien, plus générale et qui peut tout à fait s'appliquer à d'autres domaines que la programmation, est celle du MacMillan Dictionary[4] :

> a structure that supports something and makes it a particular shape
> – *MacMillan dictionary*

Dans le prochain chapitre, nous allons généraliser ce que nous avons appris jusqu'ici pour en faire un framework.

4. http://www.macmillandictionary.com/dictionary/british/framework

17 - Migrations

Introduction

Jusqu'ici nous avons agi comme si l'application bâtie sur le framework et la base de données étaient deux entités indépendantes, qui évoluaient chacune de son coté, sans vraiment tenir compte de l'autre. C'est une possibilité et vous pouvez bâtir votre framework sur cette idée.

Toutefois, la plupart des applications web aujourd'hui utilisent une base de données créée pour l'occasion. Il est donc intéressant de faire évoluer la base de données en même temps que l'application. On peut même oublier SQL et utiliser le langage du framework pour gérer les modifications de la base de données. C'est le concept de migration.

Une première migration

Comme nous avons déjà une base de données, Sequel va nous permettre de créer une migration qu'on pourra utiliser pour répliquer notre base existante, par exemple sur Heroku. Pour cela nous utiliserons le programme `sequel` avec l'option `-d` :

```
sequel -d chaine_de_connexion
```

Vous pouvez copier/coller la chaîne de connexion. Mais puisque cette chaîne réside dans le fichier `db/configuration`, autant se servir du shell. La commande suivante devrait fonctionner avec la plupart des shells :

Afficher le contenu de la base comme une migration

```
$ cat db/configuration | xargs sequel -d
```

Avec Bash, j'aurais tendance à utiliser la commande suivante, qui fait la même chose :

Une autre façon de faire, avec Bash

```
$ sequel -d $(cat db/configuration)
```

Avec Fish, le shell que j'utilise au quotidien, la commande est encore plus simple :

Pareil, mais avec Fish

```
$ sequel -d (cat db/configuration)
```

Quelque soit la méthode que vous allez choisir, elle va afficher dans le terminal le contenu de notre première migration avec Sequel :

Contenu de la base, d'après Sequel

```
$ sequel -d postgres://framework:password@localhost:5432/framework_blog
Sequel.migration do
  change do
    create_table(:posts) do
      primary_key :id
      String :title, :text=>true
      String :content, :text=>true
      DateTime :date
    end
  end
end
```

Pour conserver les migrations, rangeons les dans un nouveau dossier :

Création du dossier db/migrations

```
$ mkdir db/migrations
```

Chaque migration devra être placée dans un fichier Ruby nommé d'après le pattern `version_-description.rb`. La description pourra être le texte qui vous plaira mais la version devra répondre à un schéma précis. Soit elle sera un numéro d'ordre, comme `001`, `002`, `003`, etc. Soit elle sera une date, comme `20170401`, `20170412`, `20170423`, etc. Soit elle sera un ensemble *date + time*, comme `20170401120035`, `20170401173302`, etc. Il faudra veiller à ne pas mélanger les schémas. Le plus simple étant le numéro d'ordre, c'est celui que nous allons utiliser. Écrivez votre première migration dans un fichier `01_create_posts.rb` :

Fichier Ruby contenant la migration

```ruby
# Fichier db/migrations/01_create_posts.rb
Sequel.migration do
  change do
    create_table(:posts) do
      primary_key :id
      String :title, :text=>true
      String :content, :text=>true
      DateTime :date
    end
  end
end
```

Maintenant que nous avons une belle migration toute neuve, nous allons apprendre à l'utiliser en local. Supprimez la table `posts` de la base de données. N'hésitez pas à ouvrir deux consoles en même temps, une pour psql et une autre pour sequel. En SQL on supprime une table et son contenu avec `drop table` :

Supprimer une table avec SQL

```sql
drop table posts;
```

Avec l'utilitaire `sequel`, on lance les migrations avec l'option `-m` :

```
sequel -m dossier/des/migrations chaîne_de_connexion
```

Ce qui, avec le shell Bash, nous donnera la commande suivante :

Lancer la migration avec sequel

```
$ sequel -m db/migrations/ $(cat db/configuration)
```

Lorsque vous lancerez cette commande, ne soyez pas surpris si elle n'affiche rien. Ce sera parfaitement normal et signifiera que tout s'est bien déroulé. Vous pourrez alors regarder le contenu de votre base avec la commande `\d` dans `psql`. Vous devriez y voir deux tables (plus une *sequence* qui est en gros la description de l'incrémentation de la clé primaire de la table posts) :

```
\d
              List of relations
 Schema |     Name      |   Type   |   Owner
--------+---------------+----------+-----------
 public | posts         | table    | framework
 public | posts_id_seq  | sequence | framework
 public | schema_info   | table    | framework
(3 rows)
```

La table `schema_info` est ajoutée par Sequel pour gérer les migrations et contient la « version actuelle », c'est à dire le numéro d'ordre de la dernière migration effectuée.

Voir le contenu de la table schema_info

```
framework_blog=# select * from schema_info;
 version
---------
       1
(1 row)
```

Ajouter une seconde migration

Je voudrais ajouter un gif optionnel aux articles. Pour cela je vais créer une migration avec la version `2` et le nom `add_gif_to_posts`. Cette migration utilisera la méthode `add_column` pour ajouter la colonne `gif` à la table `posts`. Le gif sera représenté par du code HTML, comme celui que nous fourni Giphy[5]. Voici un exemple de code HTML que fourni ce service :

Code pour afficher un GIF venant de Giphy

```
<iframe src="//giphy.com/embed/tyxovVLbfZdok" width="480" height="301.2" frameBo
rder="0" class="giphy-embed" allowFullScreen></iframe> <p> <a href="https://giph
y.com/gifs/movie-happy-excited-tyxovVLbfZdok">via GIPHY</a> </p>
```

Nous aurons donc besoin d'une colonne de type *text*, et nous pouvons consulter la liste des types pour les migrations dans la documentation de Sequel[6]. Ceci nous conduit à écrire la migration suivante :

5. https://giphy.com/
6. http://sequel.jeremyevans.net/rdoc/files/doc/schema_modification_rdoc.html#label-Column+types

Une seconde migration

```ruby
# Fichier db/migrations/02_add_gif_to_posts.rb
Sequel.migration do
  change do
    add_column :posts, :gif, String, :text=>true
  end
end
```

Il faut afficher ce nouveau champ dans nos formulaires :

Un champ pour le GIF

```erb
<!--  Fichier views/shared/form_fields.html.erb -->
<label>Titre
  <input type="text" name="title" id="title" value="<%= @post.title %>">
</label>

<label>Contenu
  <input type="text" name="content" id="content" value="<%= @post.content %>">
</label>

<label>Gif
  <input type="text" name="gif" id="gif" value="<%= h(@post.gif) %>">
</label>
```

La méthode `h` permet d'afficher le code HTML comme du texte (*faites l'essai avec un formulaire de mise à jour sans cette méthode pour vous rendre compte de son utilité*). La méthode `h` est fournie par le module ERB ::Util et nous devons donc l'inclure dans notre framework :

On utilise le module ERB ::Util

```ruby
# Fichier lib/base_controller.rb
class BaseController
  # ...
  include ERB::Util
  # ...
end
```

Il faut maintenant que le contrôleur sache quoi faire de ce nouveau champ. Il est donc nécessaire de modifier les méthodes `create` et `update` de la classe PostsController :

On sauvegarde le GIF dans le contrôleur

```
class PostsController < BaseController
  def create
    Post.create(title: params["title"],
                content: params["content"],
                gif: params["gif"],
                date: Time.now)
    notice("Post créé avec succès")
    redirect_to "/posts"
  end

  def update
    post = Post[params["id"]]
    post.update(title: params["title"],
                content: params["content"],
                gif: params["gif"])
    redirect_to "/posts"
  end
end
```

Puis nous devons l'afficher dans les posts avec `<%= @post.gif %>`. Si un post n'a pas de gif, la méthode `.gif` retournera `nil` et rien ne sera affiché. Dans le cas contraire cela affichera le code HTML du gif :

Affichage du GIF

```
<!-- views/posts/show.html.erb -->
<h2><%= @post.title %></h2>
<div><i><%= Time.at @post.date %></i></div>
<p><%= @post.content %></p>

<%= @post.gif %>

<p>
  <a href="/posts/delete?id=<%= @post.id %>">
    Supprimer (mais ne venez pas pleurer après !)
  </a>
</p>
```

Généralement nous ne souhaitons pas manipuler quotidiennement des programmes en ligne de commande tel que `sequel`. En Ruby nous préférons rassembler toutes les commandes possibles dans des tâches Rake[7]. Sans entrer dans les détails, Rake est l'équivalent Ruby de Make[8]. Quand il y a peu

7. https://en.wikipedia.org/wiki/Rake_(software)
8. https://en.wikipedia.org/wiki/Make_(software)

de tâches, on utilise généralement un seul fichier nommé `Rakefile`. Le fichier suivant définit une tâche nommée `db :migrate`.

Une tâche Rake pour migrer

```
1  # Fichier Rakefile
2  namespace :db do
3    desc "Run migrations"
4    task :migrate do |t|
5      puts "Migrating…"
6      require "sequel"
7      Sequel.extension :migration
8      connexion = ENV["DATABASE_URL"] || File.read("db/configuration").chomp
9      db = Sequel.connect(connexion)
10     Sequel::Migrator.run(db, "db/migrations")
11     puts "Done."
12   end
13 end
```

Ligne 6, nous chargeons la gem Sequel et ligne 7, nous chargeons la partie de Sequel qui concerne les migrations. Enfin ligne 10, nous lançons les migrations, c'est l'équivalent en Ruby de la ligne de commande `sequel -m db/migrations $(cat/configuration)`.

Pour lancer les migrations en local vous utiliserez la commande `bundle exec rake db :migrate`. Pour les lancer sur heroku la commande sera `heroku run rake db :migrate` :

Lancer la migration en local

```
$ bundle exec rake db:migrate
Migrating…
Done.
```

Lancer la migration sur Heroku

```
$ heroku run rake db:migrate
Running rake db:migrate on □ quiet-plains-59626... up, run.2190 (Free)
Migrating…
Done.
```

Le nouveau formulaire

Et hop, en local tout fonctionne bien. Mais sur Heroku, vous aurez droit à une erreur, ça marche pas… Pourquoi ? Parce que en local, votre serveur web se lance par défaut en mode **development**, alors que sur Heroku il est lancé en mode **production** (nous verrons plus en détails la notion d'environnement dans un prochain chapitre). Et en mode production, le modèle Post n'est pas « réactualisé » après une migration. Voici l'enchaînement des faits en mode production, sur Heroku :

1. Démarrage de l'application avec une base de données sans tables.
2. Chargement du modèle Post, à partir de la base de données sans tables.
3. Migrations, la base de données contient les tables.
4. Affichage d'un formulaire à partir du modèle Post « vide » de l'étape 2, d'où erreur.

La solution la plus simple consiste à redémarrer l'application Heroku après une migration à l'aide la commande `heroku restart` pour que les modèles soient rafraichis :

Redémarrer une application sur Heroku

```
$ heroku restart
Restarting dynos on ⬢ quiet-plains-59626... done
```

Maintenant vous pouvez poster des articles avec des gifs !

Le résultat en image !