Ruby on Rails

Historique

Découverte du Framework

Rappels HTTP

MVC

Params

Bonus

Historique

- La première version de RoR date de juillet 2004
- Également appelé **RoR** ou **Rails**
- Framework web libre et open-source écrit en Ruby
- Le framework a été extrait de **Basecamp**, un outil de gestion de projets développé par *David Heinemeier Hansson*
- La version actuelle est la 7.0

RoR - Un framework qui a marqué le web

- Facilité de prototypage et rapidité d'exécution (humaine) => Popularité
- Rails a marqué le monde du développement web, et la création de frameworks web "Rails-like" (CakePHP, Symfony, CodeIgniter, ...).
- On peut produire une API, un site statique, un site dynamique, une SPA, ...
- Soulevez un rocher, trouvez une start-up basée Rails dont le site web est en .io
- Utilisé par de gros acteurs pour faire des sites webs, des micro-services, des API gateways, des monolithes.
    - Github, Twitch, Zendesk, Airbnb, Doctolib, Kickstarter, Netflix, ...

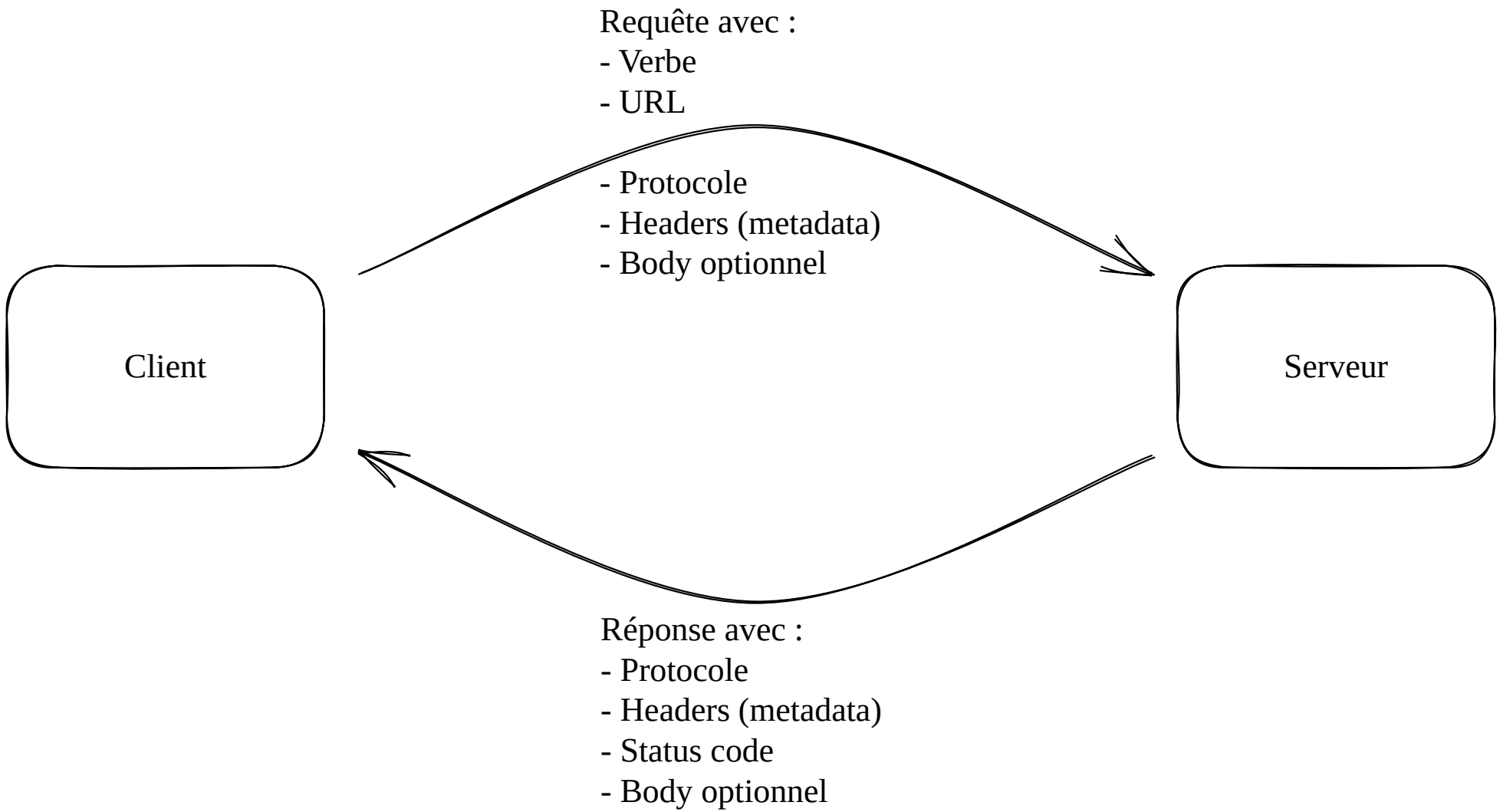
Principes / Utilisation

- Convention over configuration (CoC)
- Don't repeat yourself (DRY)
    - Rails propose des comportements par défaut pour la plupart de ses fonctionnalités
    - Qui marchent bien dans 90% des cas
- Active-Record (ORM)
- Toutes les bibliothèques : `/Active-\w+/`
- Les fichiers de configuration sont au format YAML.
- Distribué sous forme de gem : `gem install rails`

Petit rappel ~~HTTP~~ Client/Serveur Request/Response basé TCP

1. Le client établit la connexion TCP avec le serveur à une adresse donnée sur un port
2. Le client et le serveur peuvent émettre des messages (bytes).
3. Le client écrit une requête sur la socket du serveur et attend une réponse.
4. Le serveur traite la requête et répond avec une réponse.
   La réponse contient souvent un code de statut/d'erreur.
5. Le client émet de nouvelles requêtes et ça recommence.
6. Le client ou le serveur coupe la connexion.

Petit rappel HTTP



Exemple de requête HTTP

```
GET /hello.html HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

- Verbe : `GET`
- URL: `/hello.html`
- Protocole : `HTTP/1.1`
- Headers : `User-Agent`, `Host`, `Accept-Language`, `Accept-Encoding`, `Connection`
- Pas de body pour les requêtes `GET`

Exemple de réponse HTTP

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Le body arrive après les headers et deux `\n` et contient ici la page HTML.

Verbes HTTP

Method	Description
GET	Retrieve information from the server.
HEAD	Same as GET, but transfers the status line and header section only.
POST	Send data to the server for processing.
PUT	Store the body of the request on the server.
DELETE	Remove a document from the server.
TRACE	Trace the message through proxy servers to the server.
OPTION	Determine what methods can operate on a server.
CONNECT	Converts the request connection to a transparent TCP/IP tunnel.
PATCH	Applies partial modifications to a resource

Utilisation des headers HTTP

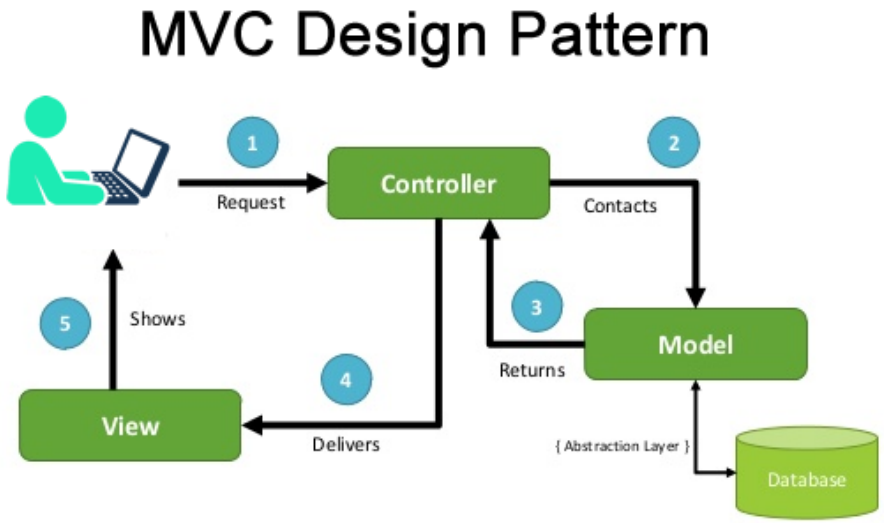
- Les headers servent à ajouter des métadonnées à la discussion
- Sert à ajouter de la négociation de contenu

  > "Voici une requête et d'ailleurs, j'accepte uniquement du JSON et XML."

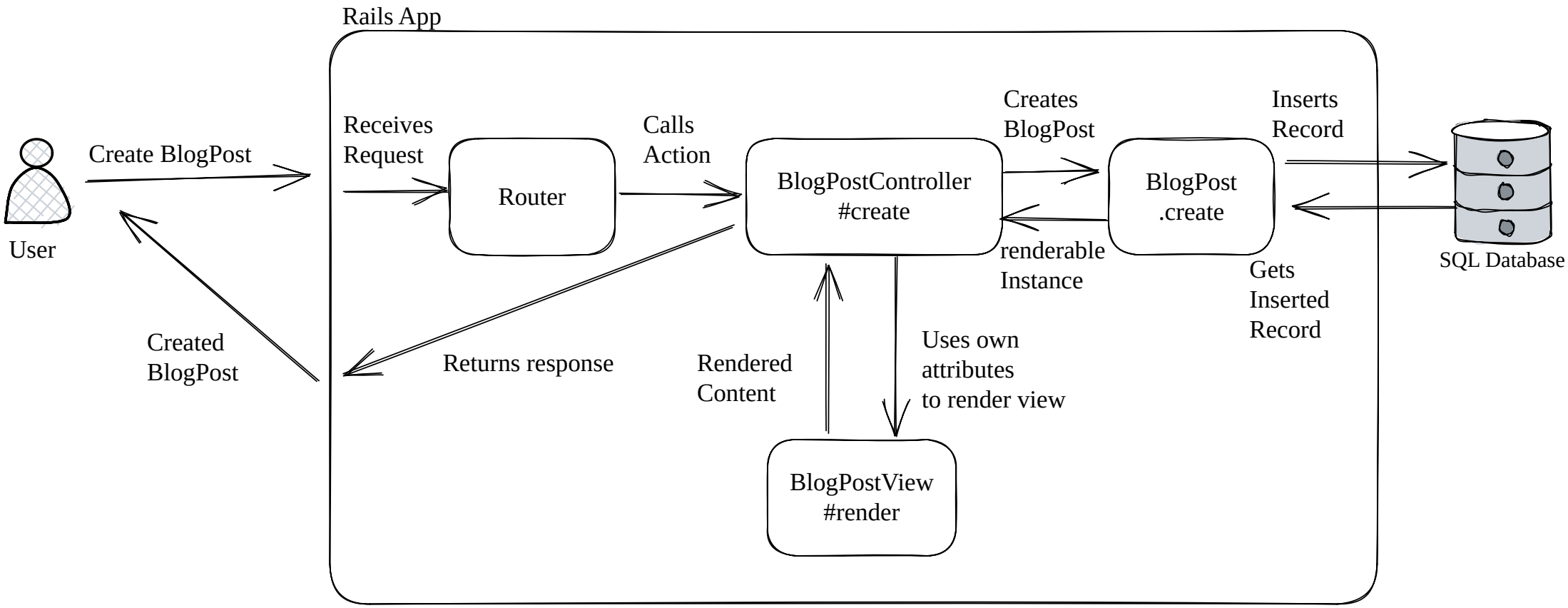
- Sert à valider avec qui on discute (user-agent, host, ...)
- Sert à valider quelle page WEB effectue une requête HTTP (CORS)
  - Vous ne préférez pas que le JS d'un site de streaming ☠️ effectue des requêtes sur le site de votre banque 💸💸💸.
- Contient les cookies
- Sert à une infinité de choses, dont des fonctionnement que **vous** pouvez ajouter

Modèle MVC

- Un **modèle (Model)** contient les données et la logique en rapport avec les données (validation, lecture et enregistrement)
- Une **vue (View)** fait l'interface avec l'utilisateur. Elle affiche les données provenant du modèle, et reçoit les actions de l'utilisateur (formulaire web, boutons, ...)
- Un **contrôleur (Controller)** fait le lien entre le modèle et la vue. Il contient la logique concernant les actions effectuées par l'utilisateur. Il modifie les modèles.



MVC sur Rails - La vie d'une requête de gestion d'article de blog



Anatomie d'un projet rails fraîchement généré

```
❯ et -l 1 -s name
my_blog (40.83 KB)
├─ Gemfile (2.28 KB)      # Gestion de dépendance bundler (TP)
├─ Gemfile.lock (5.60 KB) # Gestion de dépendance bundler (TP)
├─ README.md (374.00 B)
├─ Rakefile (227.00 B)    # Gestion des tâches de dev sur votre app (Rake)
├─ app (3.27 KB)          # Contient l'application (next slide)
├─ bin (4.28 KB)          # Contient les executables de l'application
├─ config (18.31 KB)      # Config: Routes, boot de l'app, serveur web, etc
├─ config.ru (160.00 B)   # Config pour utiliser l'app dans un serveur web
├─ db (374.00 B)          # Gestion de la DB: Schéma, migrations, etc
├─ lib                    # Bibliothèques, tâches Rake, Assets
├─ log                    # Journaux de l'application
├─ public (5.16 KB)       # Fichiers statiques à servir (favicon, robots.txt...)
├─ storage                # ...
├─ test (791.00 B)        # Tests `minitest` de votre app
├─ tmp                    # ...
└─ vendor                 # Fichiers vendorés (e.g. libs js)
```

Anatomie de l'application

```
❯ et -l 1 -s name app
app (3.27 KB)
├─ assets (864.00 B)      # Fichiers statiques de l'app (css, fonts, img)
├─ channels (164.00 B)    # Abstraction rails pour les Websockets
├─ controllers (57.00 B)  # Contrôleurs de l'application
├─ helpers (29.00 B)      # Code commun (pour les vues, etc)
├─ javascript (1.12 KB)   # Javascript de l'application (e.g. pour SPA)
├─ jobs (269.00 B)        # Tâches à réaliser en asynchrone
├─ mailers (102.00 B)     # Mailers : Sorte de contrôleur à e-mail
├─ models (74.00 B)       # Modèles ORM Active-Record
└─ views (592.00 B)       # Vues de l'app (pages et emails)
```

- Chaque chose a sa place et ça laisse peu de place au choix personnel
- Moins de débat => plus de features

Rails Router : Relier les URLS au code

- Le routeur décode chaque URL en fonction d'un schéma défini et appelle l'action du contrôleur qui correspond.
- Chaque schéma d'URL est appelé `route`.
- Les routes sont définies dans le fichier `config/routes.rb`

```
Rails.application.routes.draw do
  # on definit ici les routes de l'application
end
```

- Le bloc de code passé à `Rails.application.routes.draw` dispose d'un DSL permettant de définir facilement les routes.

Rails Router : Relier les URLS au code (2)

Quelques exemples :

```
# Définit l'action 'welcome' de DashboardController comme url racine (/)
root to: 'dashboard#welcome'

# Appelle l'action 'index' de UsersController en réponse à GET /users
get "/users", to: "users#index"

# Appelle l'action 'new' de TrainingCoursesController en réponse à GET /register
get "/register", to: "training_courses#new"

# Appelle l'action 'create' de TrainingCoursesController en réponse à POST /register
post "/register", to: "training_courses#create"
```

Dans un terminal `bin/rails routes` affiche toutes les routes qui sont définies dans l'application.

Rails Router : Récupération de paramètres depuis l'URL

- Si une action d'un contrôleur a besoin d'un ou plusieurs paramètres, on peut toujours les passer en query string (`/url?param1=value&param2=value`)
- Mais on peut définir une route qui extrait ces paramètres depuis l'URL
- Pratique pour les paramètres obligatoires

```ruby
# Appelle l'action 'show' de UsersController en réponse à GET /users/XXX
# en passant un param 'id' avec comme valeur 'XXX'
get "/users/:id", to: "users#show"
# On crée une route similaire pour GET /user-XXX
get "/user-:id", to: "users#show"

# GET /users appelle l'action 'index'
get "/users", to: "users#index"
```

Rails Router : Paramètres facultatifs

- On peut aussi ajouter des paramètres facultatifs

```
# GET /students/ accepte deux paramètres facultatifs 'year' et 'group'
get "/students/(:year)/(:group)", to: "students#index"
```

Routes REST

- Gestion de ressources (Livres, fruits, patates, comptes, etc)
- On va avoir un format d'URL dédié et une utilisation des verbes HTTP spécifique.
- Le format est souvent CRUD + L
- Create : `POST /books`
  - L'URL est la racine de la ressource
  - `POST` peut contenir un body
- Read : `GET /books/:id`
  - L'URL est paramétrisée avec l'ID de la ressource
  - `GET` ne contient pas de body

Routes REST bis

- Update : `PUT /books/:id`
  - L'URL est paramétrisée avec l'ID de la ressource
  - `PUT` peut contenir un body
  - On fera souvent un update partiel (juste les valeurs qui changent)
- Delete : `DELETE /books/:id`
  - L'URL est paramétrisée avec l'ID de la ressource
  - `DELETE` ne contient pas de body
- List : `GET /books`
  - L'URL est à la racine de la ressource
  - On s'en sert pour lister les ressources d'un type

Active Record : Implémente les modèles dans RoR

- Object-Relational Mapping (ORM)
- Chaque classe `ActiveRecord` représente une table dans la base
- Chaque instance `ActiveRecord` représente une ligne de cette table.
- Chaque instance récupère ses données depuis la base ; quand un objet est mis à jour, la ligne correspondante en base de donnée est mise à jour aussi.
- La classe implémente des accesseurs pour chaque attribut (via `ActiveRecord::Base`)

```
class Message < ApplicationRecord
end
```

- Chaque modèle hérite de `ApplicationRecord` (qui hérite de `ActiveRecord::Base`)

Action Controller : Implémente les controllers dans RoR

- Chaque contrôleur hérite de `ApplicationController` (qui hérite de `ActionController::Base`)
- Le routeur détermine quelle action du contrôleur appeler en fonction de l'url et du verbe
- Chaque action du contrôleur est une méthode d'instance
- Seules les méthodes publiques peuvent être appelées en tant qu'action

```ruby
class DashboardController < ApplicationController
  def welcome
    @title = "Bienvenue #{current_user.name}!"
    @messages = Message.all.order(date: :desc)
  end
end
```

Action View : Implémente les vues dans RoR

- Chaque action du contrôleur a une vue dédiée par défaut dans `app/views`
- Templates HTML (erb par défaut), mais possible en slim, haml
- Builder : pour 'coder' la vue en ruby directement (XML/JSON)
- Partials : pour afficher un élément qui se répète, pour simplifier les vues
- Helpers : pour formater les données (dates, monnaies), gérer les langues, les formulaires

```
# app/views/dashboard/welcome.html.erb
<h1><%= @title %></h1>
<ul>
<% @messages.each do |message| %>
    <li><%=message.body %></li>
<% end %>
</ul>
```

MVC : Convention de nommage

- **Modèles au singulier** : Le modèle `Message` (au singulier) gère la table `messages` (au pluriel) de la base de données.
- **Contrôleur au pluriel** : Le contrôleur `MessagesController` (au pluriel) gère les `Message`
- **Vues** : Chaque action d'une contrôleur à un fichier vue portant le nom de l'action, dans un dossier correspondant au nom du contrôleur au pluriel : `app\views\messages\welcome.html.erb`

⚠️ C'est parfois trompeur si on maîtrise mal le pluriel en l'anglais.

Exemples : person/people, child/ children, fish/fish, etc ...

Auto Loading

- Les applications Rails n'utilisent pas `require` ou `require_relative` pour charger le code de l'application venant d'autre fichiers ou de gems.
- Un contrôleur hérite de `ApplicationController`, mais il n'est utile de `require 'application_controller'` dans le fichier qui le définit.
- Les classes et les modules de l'application sont disponibles partout, vous n'avez pas besoin et ne devez pas utiliser `require`.

On a besoin de `require` que pour deux cas d'utilisation :

- Pour charger des fichiers depuis le dossier `lib`.
- Pour charger des gems pour lesquelles est spécifié `require: false` dans le `Gemfile`.

Params

- Le contrôleur rend accessible les paramètres envoyés de la requête dans la variable `params`
- Params s'utilise comme un dictionnaire (`Hash`)
- Toutes les valeurs reçues dans params sont de type `String`

```ruby
class CalculationsController < ApplicationController
   # get "/word-length/:word, to: 'calculation#word_length'
  def word_length
    @result = params[:word].size
  end
end
```

Params

- Rails ne fait pas la distinction entre les paramètre reçus en GET ou POST

```ruby
class CalculationsController < ApplicationController
  # post "/circle-surface/:radius, to: '#calculations#circle_surface'
  def circle_surface
    radius = params[:radius].to_i
    @result = Math::PI*(radius*radius)
  end
end
```

- Params a des méthodes en plus comme pour ajouter des validations de paramètres et parser les valeurs en types plus spécifiques (`Array`, `Hash`, `Symbol`, `Date`, `Time`, `DateTime`, ...). On verra ça plus tard.

Params - Récupérer un tableau de valeurs

`GET /sum?values[]=5&values[]=2&values[]=19`

```
class CalculationsController < ApplicationController
  def sum
    # params[:values] # => ["5", "2", "19"]
    @result = params[:values].map(&:to_i).sum
  end
end
```

L'URL sera en pratique encodée mais Rails décodera automatiquement.

```
/sum?values%5b%5d=5&values%5b%5d=2&values%5b%5d=19
```

Params - Récupérer un hash de valeurs

```
<form accept-charset="UTF-8" action="/clients" method="post">
  <input type="text" name="client[name]" value="Acme" />
  <input type="text" name="client[phone]" value="12345" />
  <input type="text" name="client[address][postcode]" value="12345" />
  <input type="text" name="client[address][city]" value="Carrot City" />
</form>
```

Lorsque ce formulaire est soumis, la valeur de `params[:client]` sera :

```
{ "name" => "Acme", "phone" => "12345", "address" => { "postcode" => "12345", "ville" => "Carrot City" } }
```

Notez les clés imbriquées dans `params[:client][:address]`.

L'objet `params` est un `Hash` mais peut accèder aux clés via des symboles ou des chaînes de caractères.

Params - Récupérer du JSON

- Si on développe un web service, on peut avoir besoin de récupérer des paramètres qui sont envoyés au format `JSON`.
- Si le header `"Content-Type"` de la requète est `"application/json"`, Rails va convertir automatiquement le `JSON`, et on pourra y acceder sous forme de Hash dans `params`.

```
curl -X POST http://localhost:3000/client/create
   -H 'Content-Type: application/json'
   -d '{ "client": { "name": "Acme", "address": "123 Carrot Street" } }'
```

La valeur de `params[:client]` sera :

```
{ "name" => "Acme", "address" => "123 Carrot Street"} }
```

Créer une application rails

Prérequis :

- ruby (>= 2.7.0), 3.0.x recommandé
- sqlite3 (et présent dans son PATH)

On installe rails (c'est une gem) :

```
gem install --user-install rails
```

On crée son application :

```
rails new doctolib
```

Cela créé une application Rails appelée doctolib dans le répertoire doctolib et installe les dépendances.

`rails new --help` affiche toutes les options du générateur d'application Rails (il y en a beaucoup !)

Génération automatique - Scaffolding

Et si maintenant je vous disais qu'il existe une fonction pour générer en une commande :

- Contrôleur
- Modèle
- Mise à jour de la DB
- Vues associées

```
$ rails generate scaffold BlogPost # Générer un nouveau MVC de BlogPost
...
# Magic is happening here...
```

On verra ça en détail plus tard.