Qui suis-je ?

- Marc DÉBUREAUX
- Master IAGL à l'université Lille 1 (2009)
- Consultant nouvelles technologies - Alten Nord

Marc DEBUREAUX

Ruby

A la découverte d'un langage libre...

C’est quoi Ruby ?

- Langage de programmation libre interprété
- Inventé par Yukihiro Matsumoto en 1995
- Dernière version 1.9.2 sortie le 19 août 2010
- Porté sur de nombreux systèmes d’exploitation

Philosophie(s)

Flexible : permet de redéfinir à loisir le coeur du langage

- Conçu pour une productivité accrue et amusante
- « Principe de moindre surprise »
- Fortement orienté-objet :
  - Toute donnée est un objet
  - Toute fonction est une méthode
  - Toute variable est une référence à un objet
- Flexible et ne contraignant pas l'utilisateur

Ruby, c'est populaire !

Liste de diffusion la plus populaire : Ruby-Talk

- Un engouement accru depuis 1995
- Les (nombreuses) conférences Ruby affichent complet
- La liste de diffusion officielle atteint + de 200 msg / jour
- 10ème langage le plus utilisé au monde (TIOBE)
- Essor motivé par des frameworks comme Rails
- Totalement libre, son développement est assuré

Ruby, c'est simple !

- Le classique «Hello World!»
  - En Java :

```java
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

  - En C :

```c
#include <stdio.h>
int main(void) {
  printf("Hello World!");
  return 0;
}
```

```
puts "Hello World!"
```

Ruby, c’est simple !

- Simple... car tout est objet !

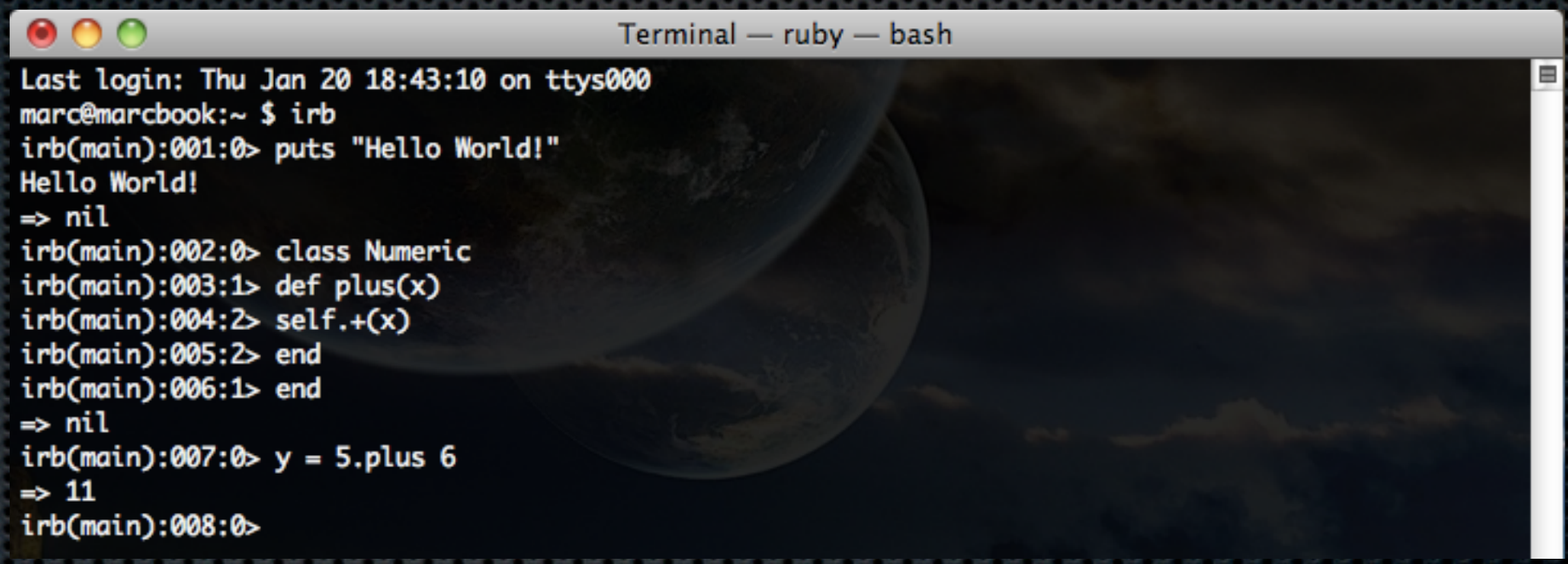
```ruby
5.times { puts "We love Ruby !" }
```

- Simple... car c’est flexible !

```ruby
class Numeric
  def plus(x)
    self.+(x)
  end
end

y = 5.plus 6
# y = 11
```

Tester Ruby ?

- Interactive Ruby : idéal pour tester le langage !



Ruby vs Java

Similitudes

- Mémoire gérée par le garbage collector
- Objets fortement typés
- Méthodes publiques, privées et protégées
- Outils de documentation (ri, rdoc) similaires à Javadoc

Différences

- Pas de compilation de code
- «end» pour clore un bloc (classe, methode, boucle, etc...)
- **Tout** est objet, même les nombres
- Pas de vérification de typage statique
- Pas de cast, que des conversions
- Le constructeur appelle «initialize»
- L'héritage multiple est possible
- Parenthèses des appels de méthodes souvent optionnelles
- «require» à la place de «import»
- «new» est une méthode et non pas un mot clé
- Pas de déclaration de type, les variables ne sont que des étiquettes

Ruby vs PHP

Tableaux et hashs identifiques sur le fond mais pas la forme

Pas besoin d'une classe de réflexion en Ruby

PHP : 0, array() == false

heredocs : «Here Document» avec <<TEXT

Typage fort : conversion implicite des types (à travers des méthodes)

PHP : abs(-1) Ruby : -1.abs

Similitudes

- Typage dynamique
- «eval» est toujours présent
- Interpolation dans les chaînes de caractères
- «heredocs» existe aussi
- Tableaux et hashs

Différences

- Typage fort
- Types primitifs tous objet
- Parenthèses souvent optionnelles
- Réflexion intrinsèque des objets
- Variables références des objets
- Tableaux et hashs non interchangeables
- Seuls «nil» et «false» valent faux
- Tout est appel de méthode en Ruby

Ruby vs Python

Documentation :
ri = pydoc

Similitudes

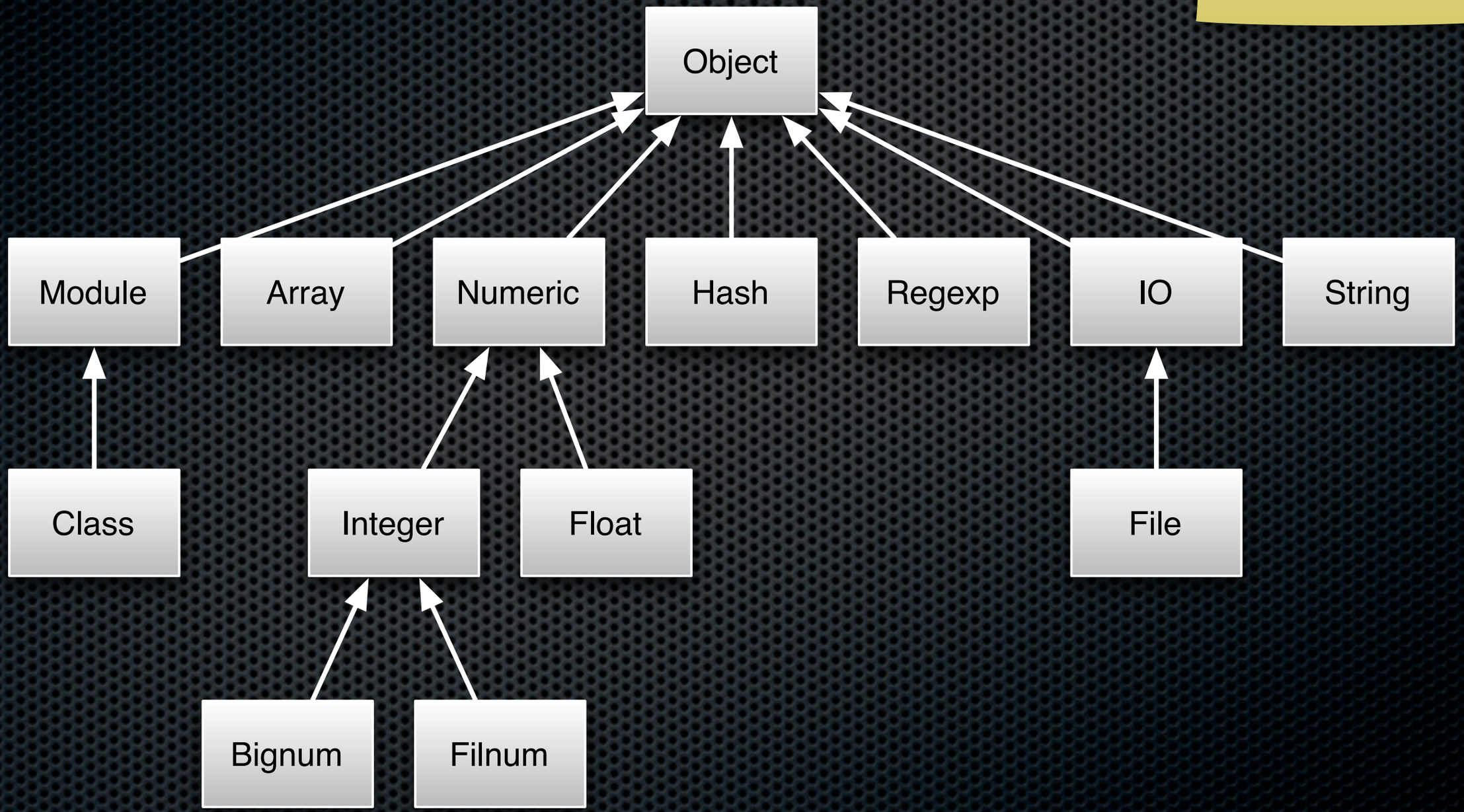
- Shell interactif
- Documentation accessible par le terminal
- Pas besoin d'un signe particulier pour marquer la fin d'une ligne
- Tableaux et hash (dict) identifiques
- Objets fortement et dynamiquement typés
- Tout est objet
- Mécanisme d'exception similaire

Différences

- Les chaînes sont altérables
- Constantes
- Règles de nommage imposées
- Seul le tableau est altérable
- L'accès aux attributs de classe se fait toujours par des méthodes
- Parenthèses souvent optionnelles
- Mots clés «public, private, protected» au lieu de «_»
- Possibilité de ré-ouvrir une classe existante pour la modifier
- true = True, false = False, nil = None
- «require» à la place de «import»
- «elsif» au lie de «elif»
- Seuls «false» et «nil» valent faux.

Les types

Superclass Kernel pour les méthodes de base comme «puts»



Généralités

Chaînes de caractères : seuls les guillemets doubles permettent l'interpolation

- Les variables sont en minuscule et peuvent contenir des chiffres et des underscores.
- Les constantes commencent par une majuscule.
- Les nombres peuvent s'écrire de plusieurs manières : 3.14 , -1.23 , 4.02e-02 , 1_000_000_000
- Les chaînes de caractères sont entourées par des guillemets simples ou doubles.

Généralités (bis)

@ = at = attribute

@@ = all attribute

Ne pas oublier $., $*, $0, $_ etc...

modules : Math par ex.

- Les variables d'instance commencent par un «@».
- Les variables de classe par «@@».
- Les variables globales par «$».
- Les symboles sont des chaînes de caractères occupant peu d'espace mémoire, mais ils ont un usage un peu particulier...
- De nombreux modules prêts à utiliser sont disponibles.

Méthodes

Interpolation :
insert le résultat d'une expression dans une chaîne de caractères

- Définir une méthode...

```ruby
def hello
    puts "Hello World!"
end

hello() # Hello World! => nil
hello   # Hello World! => nil
```

- ... avec des paramètres !

```ruby
def hello(name = "world")
    puts "Hello #{name.capitalize}!" # Interpolation
end

hello "marc"  # Hello Marc! => nil
```

Classes

```ruby
class Greeter
   def initialize(name = "World")
      @name = name
   end
   def say_hi
      puts "Hi #{@name}!"
   end
   def say_bye
      puts "Bye #{@name}!"
   end
end

g = Greeter.new("Marc")
g.say_hi  # Hi Marc! => nil
g.say_bye # Bye Marc! => nil
g.name    # Erreur!
```

- «initialize» est le constructeur.
- @name est une variable d'instance ou attribut.
- «name» est inaccessible hors de la classe ici.

Classes (suite)

- Inspecter l'instance :

```
g.inspect # => #<Greeter:0x2b88ad8 @name="Marc">
```

- Inspecter les méthodes :

```
Greeter.instance_methods(false) # => [:say_hi, :say_bye]
```

- Interroger l'instance :

```
g.respond_to?("say_hi") # => true
g.respond_to?("name")   # => false
g.respond_to?("to_s")   # => true
```

Classes (fin)

3 accesseurs :
- lecture & écriture
- lecture uniquement
- écriture uniquement

Possible de modifier une instance avec :
def obj.meth ...

- Modifier les classes *a posteriori* :

```ruby
class Greeter
   attr_accessor :name
end
```

- Accéder aux attributs :

```ruby
g.respond_to?("name")  # => true (attr_accessor, attr_reader)
g.respond_to?("name=") # => true (attr_accessor, attr_writer)

g.name = "Eric"
g.name   # => "Eric"
g.say_hi # Hello Eric! => nil
```

Aller plus loin...

```ruby
class Greeter
   attr_writer :names
   def say_hi
      if @names.nil?
         puts "Hi #{@name}!"
      elsif @names.respond_to?("each")
         @names.each do |name|
            puts "Hi #{name}!"
         end
      end
   end
end

g.say_hi # Hello Marc! => nil
g.names = ["Frank", "Eric"]
g.say_hi # => ["Frank", "Eric"]
# Hello Frank!
# Hello Eric!
```

Blocs : fonction lambda

- Les itérateurs : un moyen simple de parcourir une liste !
- Les blocs : des méthodes anonymes très fréquentes !

```ruby
@names.each { |name|
   puts "Hi #{name}!"
}
```

...avec les contrôles !

«then» permet d'écrire la condition en bloc sur une ligne

L'exemple peut être réduit à « majeur = age >= 18 »

- Conditions en bloc :

```
if/unless condition [then] # if = si vrai, unless = si faux ou nil
elsif condition [then]     # si autre condition
else                       # si aucune autre condition vrai
end
```

- Conditions multiples :

```
majeur = case
   when age >= 18 then true
   else false
end
```

- Conditions en ligne :

```
majeur = age >= 18 ? true : false # majeur = true if age >= 18
```

...avec les méthodes !

alias : copie de méthode
et non pas un lien !

* = transforme les
arguments en tableau
(doit être placé à la fin)

- Nombre variable d'arguments :

```ruby
def test(*args)
   puts args.join("-")
end

test(1, 2, 3) # 1-2-3 => nil
```

- Alias de méthodes :

```ruby
alias temp test

temp(1, 2, 3) # 1-2-3 => nil
```

...avec les ensembles !

Range : valable pour les lettres, mots et chiffres

- Déclarer un ensemble :

```
(0..10)  # ensemble de 0 à 10 inclus
(0...10) # ensemble de 0 à 9
```

- Tester un ensemble :

```
(0..10) === 3       # => true
("a".."z") === "m" # => false
```

- Utiliser un ensemble :

```
result = case score
   when 0...10 then "Inférieur à la moyenne"
end
```

...avec les tableaux !

- Opérations sur les tableaux :

```
os = ["Windows", "Gentoo", "MacOS", "BeOS", "Debian"]
linux = %w[Fedora Ubuntu Gentoo Debian]

os | linux # union de tableaux sans doublons
os & linux # intersection de tableaux sans doublons
os - linux # différence de tableaux
```

- Méthodes «bang!» :

```
os.sort  # retourne un tableau trié par ordre croissant
os.sort! # tri le tableau par ordre croissant
# même chose pour .reverse, .uniq, .rotate, etc...
```

...avec les dictionnaires

Aussi appelés «hash»

Impossible à trier...

forme json : clés uniquement symboles (préférable)

- Déclarer un hash :

```
hash = {"blue" => "color", "red" => "color", "banana" => "fruit"}
hash["red"] # => "color"
```

- Utiliser les symboles :

```
hash = {"blue" => :color, "red" => :color, "banana" => :fruit}
hash["apple"] = :color # insertion
```

- Déclaration JSON :

```
hash = {first_name: "Marc", last_name: "Debureaux"}
# => {:first_name => "Marc", :last_name => "Debureaux"}
```

...avec les blocs !

Nouvelle forme d'interpolation héritée du printf et de python avec l'opérateur %

```
class Array
   def even
      i = 0
      while i < self.size
         yield(self[i]) if i % 2 == 0
         i += 1
      end
   end
end

[1, 2, 3, 4, 5].even do |i|
   puts "%d est impair." % i
end

[1, 2, 3, 4, 5].even
# Error : no block given
```

- «yield» permet de passer la main au bloc.
- Possibilité de vérifier l'utilisation d'un bloc avec «block_given?»

...et les procs !

to_a : conversion explicite du range en tableau

& : convertir à l'appel d'une méthode seulement

```ruby
class Array
   def test(proc)
      i = 0
      while i < self.size
         proc.call(self[i])
         i += 1
      end
   end
end

proc = lambda { |i| puts "#{i}" }
(1..5).to_a.test(proc)
# test attend un objet Proc

(1..5).to_a.each(&proc)
# each attend un bloc
```

- Un objet Proc s'appelle avec la méthode «call»
- «lambda» permet de créer un objet Proc à partir d'un bloc.
- «&» permet de convertir un objet Proc en bloc

Héritage

super : inutile si aucune méthode «initialize» dans la classe fille

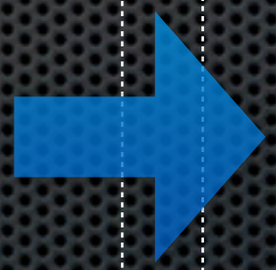
```ruby
class Animal
    def breathe
        puts "Inhale and exhale..."
    end
end

class Cat < Animal
    def speak
        puts "Meow!"
    end
end

cat = Cat.new
cat.breathe # Inhale and exhale => nil
cat.speak   # Meow! => nil
```

- L'opérateur **<** indique l'héritage.
- Il est possible de surcharger les méthodes hérités.
- Mot clé «super» pour initialiser la classe mère.

Contrôle d'accès

public	private	protected
accessible à tous	uniquement à la classe	à la classe et sous-classes

```ruby
class Test
   def method1
     # méthode publique
   end
   protected
   def method2
     # méthode protégée
   end
   private
   def method3
     # méthode privée
   end
end
```



```ruby
class Test
   def method1
     # méthode publique
   end
   def method2
     # méthode protégée
   end
   def method3
     # méthode privée
   end
   public :method1
   protected :method2
   private :method3
end
```

Exceptions

Type d'erreur non obligatoire, Exception si non fourni

return non obligatoire !

- Exemple d'utilisation :

```ruby
def inverse(x)
    raise ArgumentError, "L'argument doit être numérique"
        unless x.is_a? Numeric
    return 1.0 / x
end

begin
    puts inverse(2)     # => 0.5
    puts inverse("A")   # Erreur !
rescue ArgumentError => e
    puts e.message      # L'argument doit être numérique => nil
    puts e.backtrace.inspect  # trace d'exécution
else
    puts "Erreur non gérée"
end
```

Modules et mixins

Utile pour les constantes réutilisables

Namespace : voir fonctionnement de Math, préfixage des méthodes avec nom du module nécessaire

```ruby
module Skill
    def move
        puts "Je bouge !"
    end
end

class Animal
    include Skill

    def eat
        puts "Je mange !"
    end
end

animal = Animal.new
animal.eat  # Je mange ! => nil
animal.move # Je bouge ! => nil
```

- «module» : ne peut être instancié ni hérité !
- Partage des fonctionnalités entre les classes.
- Peut servir de namespace à des fonction de base.

That’s all folks!

Questions ?

Attention ! Ce cours est loin d’être exhaustif, n’oubliez pas de RTFM quand même...

Exercice 1

- Créer un programme permettant de reproduire le chiffrement de César.

```
c = Cesar.new "Voici le chiffre de Cesar !"
c.encode # => "Ibvpv yr puvsser qr Prfne !"
```

- Créer un programme permettant de reproduire le chiffrement de Vigenère.

```
c = Vigenere.new "Voici le chiffre de Vigenere !" "Vigenere"
c.encode # => "Qwogv pv gcqljei ui Qqmiaiii !"
```

Exercice 2

- Créer un programme pour faire des vagues...
  - ... de manière impérative !
  - ... puis de manière récursive !

```
Vagues.start 5 2 # x = hauteur de vague, y = nombre de vagues
# *
# **
# ***
# ****
# *****
# ****
# ***
# etc...
```

Marc DEBUREAUX

Ruby on Rails

Un framework web à grande vitesse !

Rails, c’est quoi ?

- Framework web libre écrit en Ruby.
- Première version (stable) sortie en 2005.
- Basé sur le pattern Modèle-Vue-Contrôleur.
- «Convention over configuration»
- «Don’t repeat yourself !»

Bonnes habitudes

- Utiliser RVM (Ruby Version Manager)
  http://rvm.beginrescueend.com/
- Utiliser un gestionnaire de version (Git, Subversion, etc.)
  http://git-scm.com/ http://tigris.subversion.com/
- Mettre en place des tests unitaires et d'intégration tôt dans l'application pour détecter la régression.
- User et abuser des gems et des bundles...
  mais on verra tout ça plus tard !

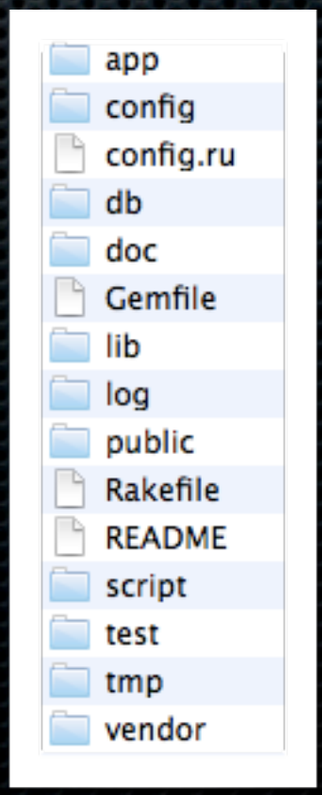
Commençons !

test/ : obsolète avec l'utilisation de rspec

- Génération du projet :

```
$ rails new tprails
```

- Architecture :
  - `app/` : coeur de l'applications (models, views, controllers, helpers...)
  - `config/` : configuration de l'application
  - `doc/` : documentation de l'application
  - `db/` : base de données et migrations
  - `public/` : pages statiques, feuilles de style, images, etc...
  - `vendor/` : plugins et gems



Préparer l'environnement

- Gems & Gemfile :

```
source 'http://rubygems.org'
gem 'rails', '3.0.3'
gem 'sqlite3-ruby', :require => 'sqlite3'
```

- Attention ! Rails sous GNU/Linux nécessite l'installation de paquets préalables.
- Bundle it !

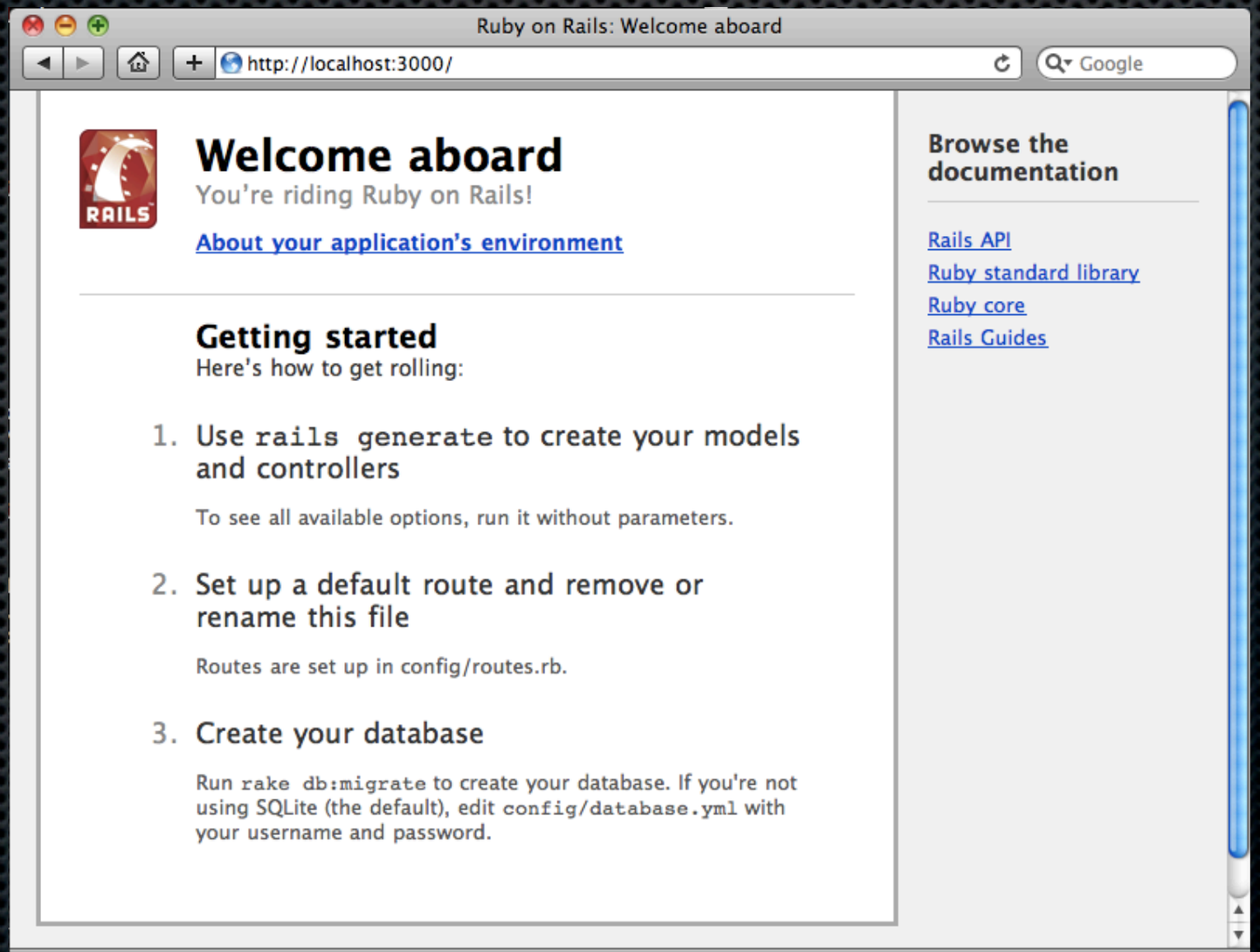
```
$ bundle install
```

- And run it !

```
$ rails server
```

Et maintenant ?

- Ça fonctionne... mais y a rien (d'intéressant) !



Contrôler ses sources

Fichiers inutiles :

.swp de vim et emacs
.DS_Store sous Mac OS X

Voir documentation de Git pour les options «branches» et «github»

- Configuration de Git :

```
$ git config --global user.name "Marc DEBUREAUX"
$ git config --global user.email "marc@debnet.fr"
$ git config --global core.editor "mate -w"
```

- Remplacez «mate -w» par votre éditeur préféré...

- Commencer à utiliser Git :

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

- Configurez plus finement le `.gitignore` pour éviter de prendre les fichiers inutiles.

Les tests avant tout !

- RSpec ou la création de tests intuitifs (et amusants ?)
- Un petit détour par le Gemfile :

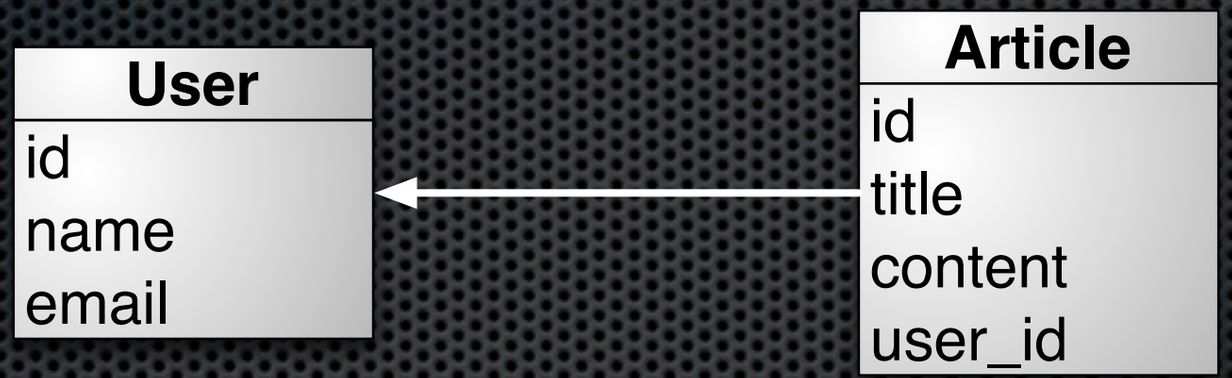
```
group :development do
   gem 'rspec-rails', '2.4.1'  # Ajoute le générateur RSpec
end

group :test do
   gem 'rspec', '2.4.0'   # Moteur d'exécution des tests RSpec
   gem 'webrat', '0.7.3'  # Tests d'intégration
end
```

- On installe le tout et ça roule !

```
$ bundle install
$ rails generate rspec:install
```

Jouer avec les modèles

- Objectif :

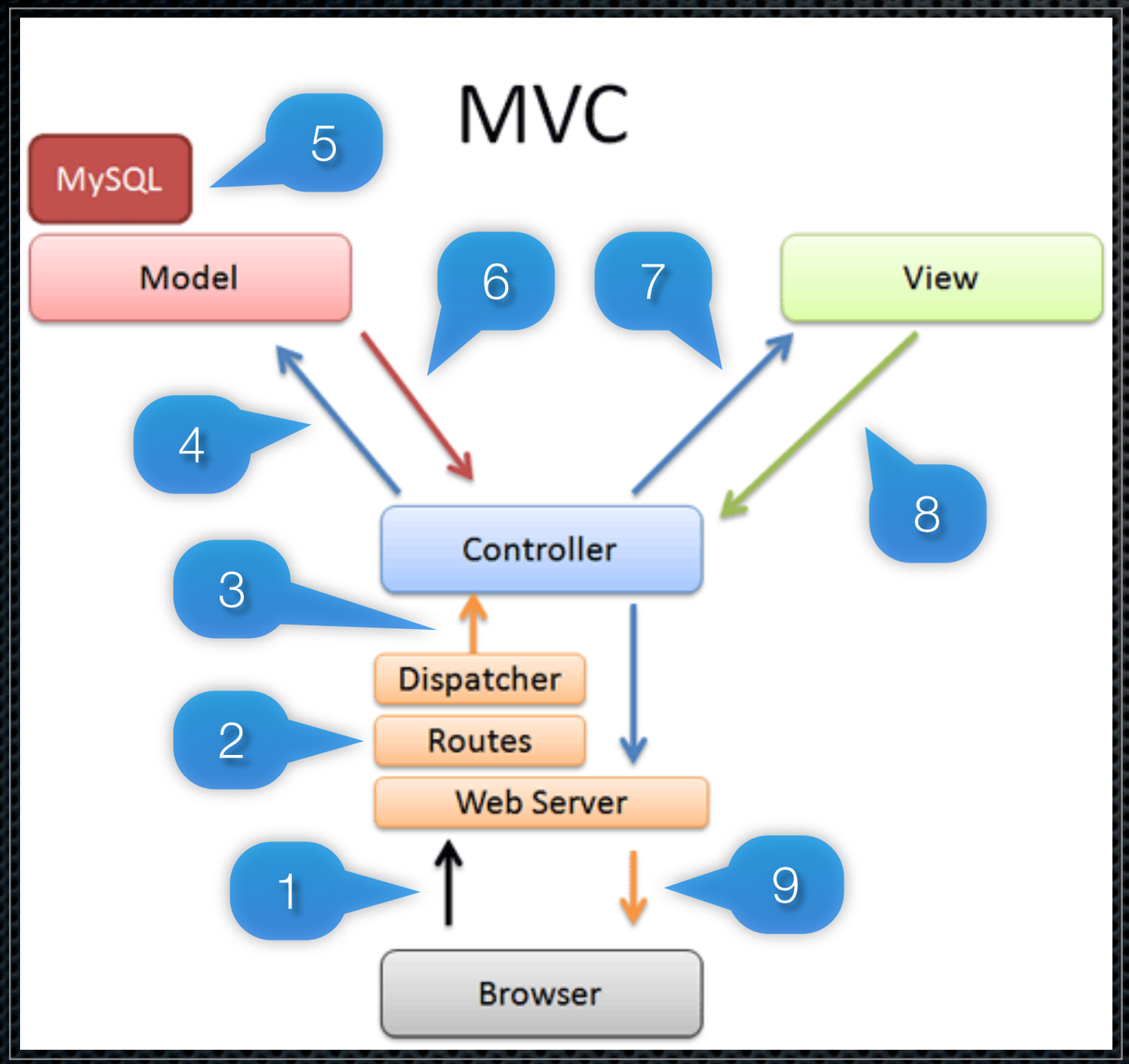


- Implémentation tout-en-un :

```
$ rails generate scaffold User name:string email:string
$ rake db:migrate
```

- Attention ! Bien que le scaffolding permette de démarrer rapidement, il est souvent nécessaire d'adapter le code généré. Les développeurs expérimentés préfèrent éviter cette méthode et tout écrire à partir de zéro.

Démonstration



1. Requête GET sur /users
2. Routage de la demande
3. Rails appelle index du contrôleur
4. Le contrôleur questionne le modèle
5. Le modèle interroge la base
6. Le modèle retourne le résultat
7. La vue est appelée avec les données
8. La vue génère le HTML
9. Le HTML est retourné au navigateur

Le poste d'aiguillage

R.E.S.T. :
REpresentational State Transfert

Composants web considérés comme des ressources CRUD

- Routage standard et RESTful dans Rails :

```
resources :users
```

Méthode	Chemin	Action
GET	/users	index
GET	/users/new	new
POST	/users	create
GET	/users/1	show
GET	/users/1/edit	edit
PUT	/users/1	update
DELETE	/users/1	destroy

C.R.U.D.

Create

```
u = User.new
u.name = "Marc"
u.save
```

Read

```
u = User.find(1)
```

Update

```
u = User.find(1)
u.name = "Marc"
u.save
```

Delete

```
u = User.find(1)
u.destroy
```

Et maintenant ?

- Faire la même chose avec les articles...

```
$ rails generate scaffold Article \
>    title:string content:text user_id:integer
$ rake db:migrate
```

- Quels sont les problèmes ?
  - Aucune validation des données
  - Aucun test (c’est mal !)
  - Aucune mise en forme
  - On comprend pas comment ça marche (ou alors vous êtes doués !)...

Valider ses données

- Multiples méthodes de validations, exemple :

```
class User < ActiveRecord::Base
   validates_presence_of :name
end
```

- Ce qui donne :

```
u = User.new  # => #<User id: nil, name: nil, email: nil>
u.save        # => false (voir u.valid?)
u.errors      # => { :name => ["can’t be blank"] }
```

- La validation du modèle impacte également le résultat web, testez-le !

Validations à la pelle

- `validates_presence_of :name`
- `validates_numericality_of :number`
- `validates_uniqueness_of :something`
- `validates_confirmation_of :password`
- `validates_acceptance_of :model`
- `validates_length_of :field, :minimum => 10, :maximum => 100`
- `validates_format_of :email, :format => /regex/i`
- `validates_inclusion_of :age, :in => 18..99`
- `validates_exclusion_of :age, :in => 0...18`

Relations simples

- Célébrons l’union de nos deux modèles :

```ruby
class User < ActiveRecord::Base
   has_many :articles  # Pluriel
end
```

```ruby
class Article < ActiveRecord::Base
   belongs_to :user    # Singulier
end
```

- Et voici le résultat :

```ruby
user = User.first  # => premier utilisateur enregistré
user.articles      # => liste des articles de l’utilisateur
```

Pages statiques

La génération du contrôleur crée :

- les tests RSpec
- les pages ERB associées
- les routes GET pour chaque action

- 2 genres de pages statiques :
  - Les «vraies» pages statiques en HTML dans /public.
  - Les «pages statiques» à la sauce Rails utilisant le pattern MVC.
- Créons un contrôleur pour gérer nos pages :

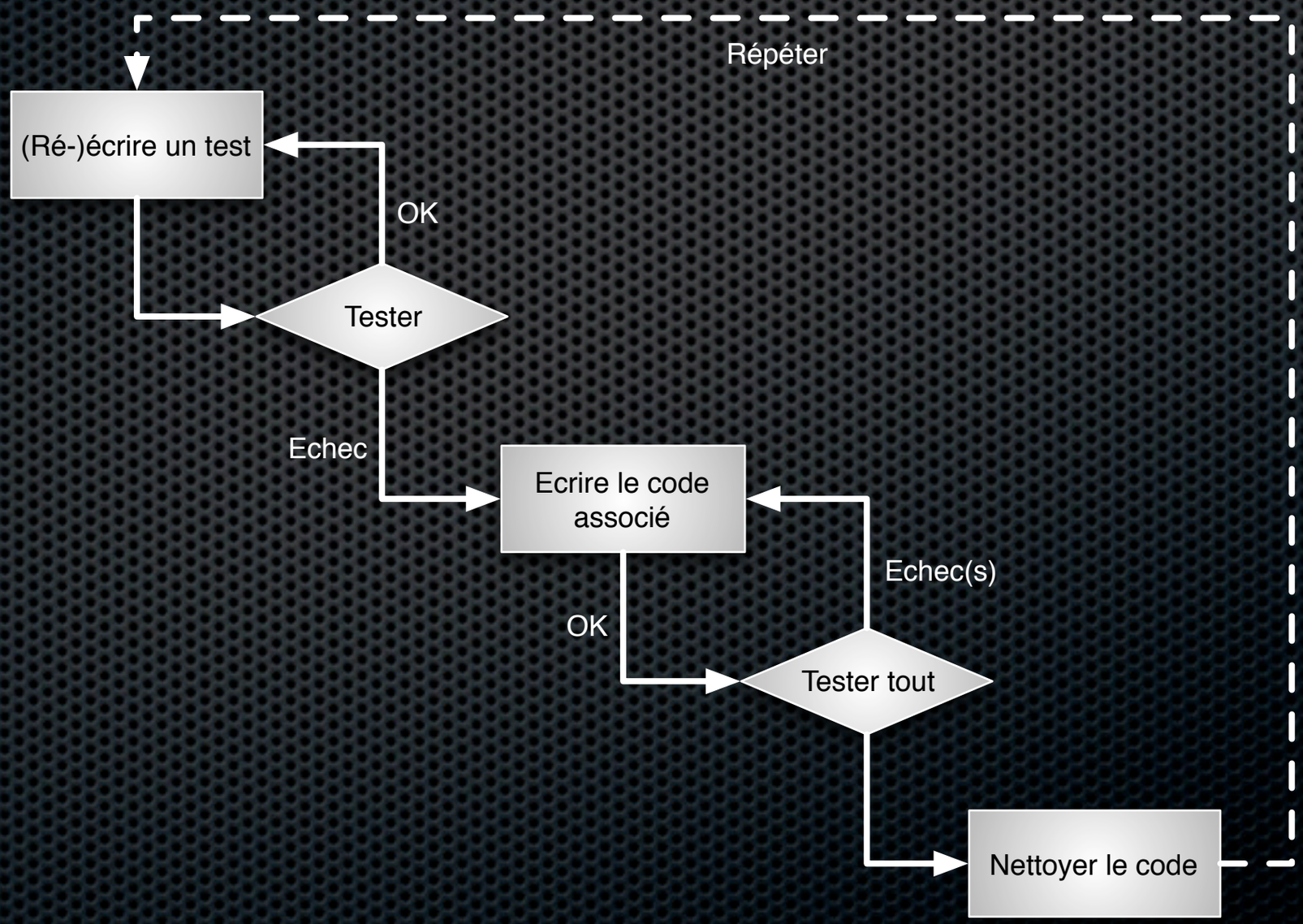
```
$ rails generate controller Pages home contact
```

- Et essayons le tout !

```
$ rails server
```

Red, Green, Refactor !

- Cycle de conception en test-driven development :



The Rails Way

Dans l'exemple, pas besoin de tests sur les vues et les helpers :

```
[git] rm -rf spec/views
[git] rm -rf spec/helpers
```

- RSpec : syntaxe dédiée à la création de tests
- Exemple d'utilisation :

```ruby
require "spec_helper"

describe PagesController do                 # Tests du contrôleur Pages
   render_views                             # Permet le rendu des vues

   describe "GET 'home'" do                 # Tests de l'action «home»
      it "should be successful" do          # Description du test
         get 'home'                         # Aller à l'action «home»
         response.should be_success         # La réponse doit être valide
      end
   end
end
```

Exercices

Ne pas oublier le

rake db:test:prepare

pour que la base de données soit accessible aux tests.

- Écrire les cas de tests pour les actions *help* et *about*.
- Faire en sorte que le test réussisse pour l'action *about*.
  - Ajouter la définition de l'action dans le contrôleur *Pages*.

```
def about
end
```

  - Ajouter la page *about.html.erb.*
  - Ajouter le routage vers l'action dans *routes.rb*.

```
get 'pages/about'
```

Quelques banalités

Il est possible de définir un layout différent dans le contrôleur

- Les layouts permettent d'avoir une base commune à toutes les pages, par défaut c'est `application.html.erb`

```
<!DOCTYPE html>
<html>
   <head>
      <title>TP Rails</title>
   </head>
   <body>
      <%= yield %>  <!-- yield insère ici la vue correspondante -->
   </body>
</html>
```

- `<% %>` interprète du code Ruby, `<%= %>` affiche son résultat

Be dynamic !

- Objectif : changer le titre de chaque page.
- Tout d’abord, préparons les tests !

```
describe "GET 'home'" do
   # [...]
   it "should have the right title" do
      get 'home'
      response.should have_selector("title", :content => "Accueil")
   end
end
```

Be dynamic ! (suite)

- Ajoutons le titre dans le contrôleur *Pages* :

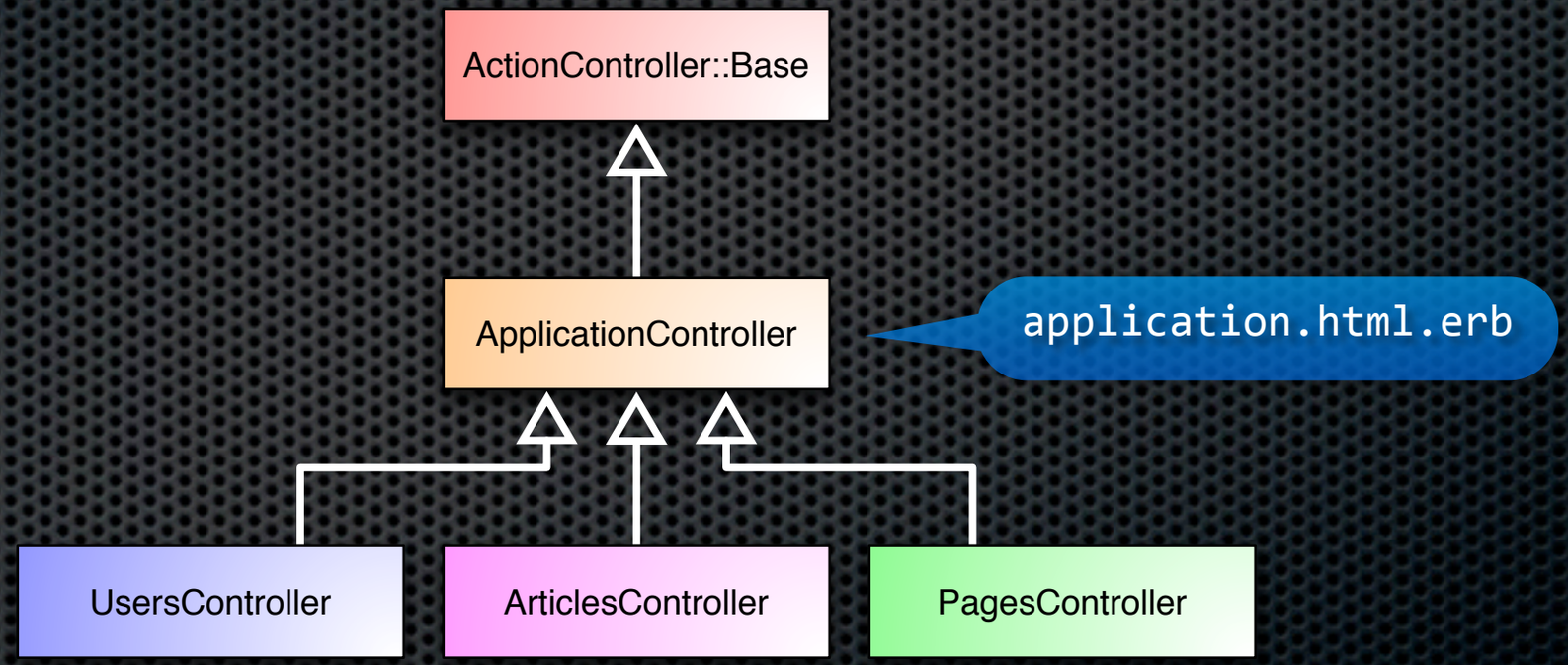
```
def home
   @title = "Accueil"
end
```

- Puis affichons-le dans le layout :

```
<!DOCTYPE html>
<html>
   <head>
      <title>TP Rails | <%= @title %></title>
   </head>
   <body>
      <%= yield %>
   </body>
</html>
```

Et maintenant ?

- Ça marche... mais pourquoi ?



- En fait, ça ne marche pas pour tout le monde !

Helpers à la rescousse

- Les helpers offrent leurs fonctions au service des vues.
- Un helper peut régler notre problème de titre :

```ruby
module ApplicationHelper
   def global_title
      base_title = "TP Rails"
      if @title.nil? then
         return base_title
      else
         return "#{base_title} | #{@title}"
      end
   end
end
```

- Il suffit ensuite d'appeler la méthode *global_title* dans le layout.

Helpers bien connus

CSRF : Cross-site request forgery.

Dans le cas de rails, les formulaires sont protégés par un jeton de validité.

- `csrf_meta_tag` : protège les formulaires des attaques de type CSRF en leur fournissant un jeton de validité.
- `stylesheet_link_tag :all` : injecte toutes les feuilles de style trouvées dans le répertoire /public/stylesheets/.
- `stylesheet_link_tag "monstyle/screen", :media => "screen"` : injecte le CSS nommé *screen.css* du répertoire /public/stylesheets/monstyle/.
- `javascript_include_tag :defaults` : injecte les javascripts standards utilisés dans Rails (framework Prototype).
- `link_to "A propos de ce site", "pages/about"` : génère un lien hypertexte vers une destination précise (routage standard)
- `image_tag "logo.png", :alt => "Logo", :class => "image"` : insère une image /public/images/logo.png avec les attributs suivants.

Diviser pour mieux régner

- Les partiels : permettent de scinder les vues.

```
<!DOCTYPE html>
<html>
    <head>
        <title>TP Rails | <%= @title %></title>
        <%= render "layouts/stylesheets" %>
    </head>
    <body>
        <%= yield %>
    </body>
</html>
```

- /app/views/layouts/_stylesheets.html.erb :

```
<%= stylesheet_link_tag :all %>
```

Tests d'intégration

Quel intérêt alors ? Simple, les tests d'intégration ne sont pas limités à la portée d'un contrôleur et peuvent interagir sur toute l'application. Idéal pour tester les routes.

- Permettent de simuler un navigateur parcourant le site.
- Génération des tests d'intégration :

```
$ rails generate integration_test main_links
```

- Ça vous rappelle quelque chose ?

```ruby
require "spec_helper"

describe "MainLinks" do
   it "should have a home page at /" do
      get '/'
      response.should be_success
   end
end
```

Liens nommés...

- Permettent de modifier le routage des liens.
- Exemple :

```
<a href="/pages/about">A propos</a> <!-- deviendrait -->
<a href="/about">A propos</a>
```

```
<%= link_to "/pages/about", about_path %>
```

- Cependant, ce n’est pas magique :
  Il faut modifier le routage dans `routes.rb` !

... pour routes nommées !

- Actuellement nous avons quelque chose comme ça :

```
TPRails::Application.routes.draw do
   get "pages/about"  # Crée une route simple (GET) vers about
end
```

- Pour que nos liens nommés fonctionnent, il faut ça :

```
TPRails::Application.routes.draw do
   match "/about", :to => "pages#about"
   # Crée une route nommée vers l'action about de PagesController
end
```

- Pour la racine de l'application :

```
root :to => "pages#home"
```

That’s all folks!

Questions ?

Je vous invite à lire la documentation sur http://guides.rubyonrails.org !

Au programme

- Migrations : la brique élémentaire de l'ORM Rails.
- Tests unitaires autour des modèles.
- Routes et ressources standards.
- Authentification simple et sécurité des données.
- Formulaires et validations.
- Et plus encore !

Migrations

Les migrations sont des fichiers Ruby préfixés par un timestamp de création pour éviter les conflits.

Faire un db:migrate pour prendre en compte les nouvelles migrations

- Permettent de modifier la base de données en Ruby :

```ruby
class CreateUsers < ActiveRecord::Migration
   def self.up                          # Installation de la migration
      create_table :users do |t|        # Création d'une table «users»
         t.string :name                 # Champ «name» type varchar
         t.string :email                # Champ «email» type varchar
         t.timestamps                   # Ajout du created_at/updated_at
      end
   end

   def self.down                        # Migration rollback
      drop_table :users                 # Drop la table «users»
   end
end
```

Bien éduquer ses modèles

- Les helpers `attr_accessible` et `attr_protected` :

```
class User < ActiveRecord::Base
   attr_accessible :name, :email
end
```

- Permettent d'éviter une vulnérabilité par rapport à l'assignement de masse

```
User.create(:name => "Marc", :email => "marc@debnet.fr")
# => #<User id: 1, name: "Marc", email: "marc@debnet.fr", ...
```

- Valider correctement ses données...
- ... et les tester convenablement !

Tester les modèles

Les tests à partir de cet exemple concernent le modèle et non plus le contrôleur :

spec/models/user_spec.rb

- Actuellement nous avons ça qui ne fait rien :

```
describe User do
   pending "add some examples to (or delete) #{__FILE__}"
end
```

- Un premier test permettant de vérifier la validation :

```
describe User do
   before :each do
      @attr = { :name => "Exemple", :email => "foo@bar.com" }
   end
   it "should require a name" do
      user = User.new @attr.merge(:name => "")
      user.should_not be_valid
   end
end
```

Exercices

- Écrire les tests et la validation pour limiter la taille du nom d'utilisateur de 6 à 30 caractères.
- Écrire les tests et la validation pour vérifier que l'adresse email est saisie et qu'elle ait le bon format.
  - Vous pouvez utiliser Rubular (http://www.rubular.com) pour vos regex.
- Écrire les tests et la validation pour empêcher la création de deux utilisateurs avec le même email.

Correction

- Écrire les tests et la validation pour limiter la taille du nom d'utilisateur de 6 à 30 caractères.

```ruby
class User < ActiveRecord::Base
   validates_length_of :name, :within => 6..30
end
```

```ruby
it "should reject too long names" do
   user = User.new @attr.merge(:name => "a" * 31)
   user.should_not be_valid
end

it "should reject too short names" do
   user = User.new @attr.merge(:name => "a" * 5)
   user.should_not be_valid
end
```

Correction

- Écrire les tests et la validation pour vérifier que l'adresse email est saisie et qu'elle ait le bon format.

```
class User < ActiveRecord::Base
   validates_presence_of :email
   validates_format_of :email, :with =>
      /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
end
```

```
it "should accept valid emails" do
   emails = %w[foo@bar.com foo@bar.foo.com foo.bar@foo.com]
   emails.each do |email|
      user = User.new @attr.merge(:email => email)
      user.should_not be_valid
   end
end
# Faire la même chose mais avec des emails invalides
```

Correction

Uniqueness insuffisant : dans le cas où l'utilisateur valide deux fois un formulaire par exemple

- Écrire les tests et la validation pour empêcher la création de deux utilisateurs avec le même email.

```
class User < ActiveRecord::Base
   validates_uniqueness_of :email, :case_sensitive => false
end
```

```
it "should reject duplicate emails" do
   User.create! @attr          # Création d'un utilisation => OK
   user = User.new @attr       # Création du même utilisateur => NOK
   user.should_not be_valid
end
```

- Attention ! La contrainte `uniqueness` n'est pas suffisante dans certains cas.

Presque unique !

Uniqueness insuffisant : dans le cas où l'utilisateur valide deux fois un formulaire par exemple

- Une nouvelle migration pour résoudre le problème :

```
$ rails generate migration add_email_uniqueness_index
```

```
class AddEmailUniquenessIndex < ActiveRecord::Migration
    def self.up
        add_index :users, :email, :unique => true
    end
    def self.down
        remove_index :users, :email
    end
end
```

- ... et une migration, une !

```
$ rake db:migrate         # Exécute les dernières migrations
$ rake db:test:prepare    # Copie la base de données pour les tests
```

Et maintenant ?

Uniqueness insuffisant : dans le cas où l'utilisateur valide deux fois un formulaire par exemple

- Vous devriez maintenant comprendre une partie des contrôleurs et vues générées par le *scaffolding*.
- Débugger les données envoyées en paramètres :

```
<%= debug(params) if Rails.env.development? %>
```

  - «params» est un hash contenant toutes les données envoyées au contrôleur.
  - *Rails.env* permet de connaître l'environnement d'exécution.

Un peu de sécurité

- Ajoutons un mot de passe à la création d'un utilisateur.
- Commençons par les tests !

```
before :each do
   @attr = { :name => "Exemple", :email => "foo@bar.com",
      :password => "foobar", :password_confirmation => "foobar" }
end

it "should require a password" do
   user = User.new @attr.merge(:password => "")
   user.should_not be_valid
end

it "should have password and confirmation matching" do
   user = User.new @attr.merge(:password_confirmation => "barfoo")
   user.should_not be_valid
end
```

… dans le modèle ?

Pas besoin de créer l'attribut password_confirmation car la validation le fait à notre place.

password est ici un attribut modèle virtuel

- Forcément, tous les tests plantent !
- Modifier le modèle pour permettre cette fonctionnalité :

```ruby
class User < ActiveRecord::Base
   attr_accessor :password
   attr_accessible :name, :email, :password, :password_confirmation
   # Validations précédentes
   validates_confirmation_of :password
end
```

- Pas de migration ? Bah non… on ne veut pas garder le mot de passe en clair dans la base de données.

... dans les données ?

- Stocker le mot de passe chiffré en base de données :

```
$ rails generate migration add_password_to_users
```

```
class AddPasswordToUsers < ActiveRecord::Migration
   def self.up
      add_column :users, :secure_password, :string
   end
   def self.down
      remove_column :users, :secure_password
   end
end
```

- ... et comme d'habitude :

```
$ rake db:migrate          # Exécute les dernières migrations
$ rake db:test:prepare     # Copie la base de données pour les tests
```

... et dans le code ?

- Ajouter une méthode de chiffrement dans le modèle :

```ruby
class User < ActiveRecord::Base
    # [...]
    before_save :encode_password   # Méthode exécutée avant save

    private
    def encode_password
        self.secure_password = encode(password)
    end
    def encode(string)
        return Digest::SHA2.hexdigest(string)   # Chiffrement SHA2
        # Nécessite require "digest"
    end
end
```

Pour aller plus loin...

- SHA2 est une méthode de hashage qui ne garantie pas à elle seule la fiabilité d'un mot de passe.

  Consultez http://fr.wikipedia.org/wiki/Table_arc-en-ciel et proposez une méthode pour améliorer la sécurité.

- Créez une méthode publique `password_ok?` dans le modèle permettant de vérifier que le mot de passe est correct, et écrivez les tests associés.

Passe-moi le sel

- Préparons une migration :

```
$ rails generate migration add_salt_to_users salt:string
```

```
$ rake db:migrate          # Exécute les dernières migrations
$ rake db:test:prepare     # Copie la base de données pour les tests
```

- ... puis dans le modèle :

```
class User < ActiveRecord::Base
   # [...]
   private
   def encode_password
      self.salt = encode("#{Time.now.utc}:#{password}")
         if new_record?
      self.secure_password = encode("#{salt}:#{password}")
   end
end
```

Authentification

- Méthode de classe permettant de récupérer un utilisateur à partir de son email et mot de passe :

```
class User < ActiveRecord::Base
   # [...]
   def password_ok?(password)
      return (secure_password == encode("#{salt}:#{password}"))
   end

   def self.authenticate(email, password)
      user = User.find_by_email(email)
      return nil if user.nil?
      return user if user.password_ok?(password)
   end
   # [...]
end
```

Inscrivez-vous !

Le formulaire de création (et modification) est un partiel avec scaffold.

- Comprendre le formulaire généré par *scaffolding* et insérer les champs manquants du modèle.

```
<!-- [...] -->
<div class="field">
    <%= f.label :password %><br />
    <%= f.password_field :password %>
</div>
<div class="field">
    <%= f.label :password_confirmation, "Confirmation" %><br />
    <%= f.password_field :password_confirmation %>
</div>
<!-- [...] -->
```

- ... et ajouter un titre à notre page via le contrôleur.

Tester les formulaires

assigns permet de récupérer la valeur d'instance pointée par le symbole dans le contrôleur

- Dans les tests du contrôleur `UsersController` :

```
describe "User creation success" do
   before :each do
      @attr = { :name => "Exemple", :email => "foo@bar.com",
      :password => "foobar", :password_confirmation => "foobar" }
   end

   it "should create a user" do
      lambda do
         post :create, :user => @attr
      end.should change(User, :count).by(1)
   end

   it "should be redirected to the user" do
      post :create, :user => @attr
      response.should redirect_to(user_path(assigns(:user)))
   end
end
```

Sessions

- Permettent de sauvegarder un état de l'utilisateur de manière semi-permanente sur le serveur.
- Il est judicieux de modéliser les sessions en RESTful :

```
$ rails generate controller Sessions new create destroy
```

- ... avec les routes qui vont bien :

```
resources :sessions, :only => [ :new, :create, :destroy ]
```

Connexion

Etant donné que le formulaire n'est pas basé sur un modèle comme pour les utilisateur, il faut définir explicitement l'accès au contrôleur dans le form_for.

- Le formulaire de connexion de l'utilisateur :

```
<%= form_for(:session, :url => sessions_path) do |f| %>
  <div class="field">
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </div>
  <div class="field">
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </div>
  <div class="actions">
    <%= f.submit "Connexion" %>
  </div>
<% end %>
```

Connexion (suite)

signin_path :
Il faut créer une route nommée vers session#new pour que ça fonctionne.

flash -> redirect_to
flash.now -> render

- ... et le contrôleur associé :

```
class SessionsController < ApplicationController
   def new   # Action pour le formulaire de connexion
      @title = "Connexion"
   end
   def create   # Authentification de l'utilisateur
      data = params[:session]
      user = User.authenticate(data[:email], data[:password])
      if user.nil? then
         flash[:error] = "Mot de passe ou email invalide !"
         redirect_to signin_path
      else
         session[:user] = user
         redirect_to root_path
      end
   end
end
```

Exercices

Le «before_filter» dans le contrôleur est bien utile pour protéger les pages.

Encore mieux si les tests sont faits avant d'écrire le code métier...

Utilisez les helpers pour factoriser les méthodes d'authentification (cf. mixins)

- Rappeler sur chaque page le nom de l'utilisateur connecté et dans le cas contraire prévoir un lien pour s'authentifier.
- Écrire la méthode correspondante à l'action `destroy` permettant à un utilisateur de se déconnecter.
- Corriger l'application pour que le lien entre les articles et les utilisateurs soit implicite et que la création d'un article soit inaccessible si non connecté.

Exercices (suite)

- Interdire à l'utilisateur connecté de modifier un autre profil que le sien, et interdire globalement la modification si non connecté.
- Améliorer la méthode de connexion pour permettre à l'utilisateur de revenir à la page qu'il avait quittée.
- Pour les plus rapides : proposer et implémenter une méthode sécurisée utilisant les cookies.

Correction n°1

- Rappeler sur chaque page le nom de l'utilisateur connecté et dans le cas contraire prévoir un lien pour s'authentifier.

```
module ApplicationHelper
   def signin_link
      user = session[:user]
      if user.nil? then
         return link_to("Se connecter", signin_path)
      else
         link = "Bonjour, #{user.name} ! "
         link << "#{link_to 'Se déconnecter',
            signout_path, :method => :delete}"
         return link.html_safe
      end
   end
end
```

```
<%= signin_link %> <!-- dans application.html.erb -->
```

Correction n°2

- Corriger l'application pour que le lien entre les articles et les utilisateurs soit implicite et que la création d'un article soit inaccessible si non connecté.

```
class SessionsController < ApplicationController
   def destroy
      session.delete :user
      flash[:notice] = "Vous avez été déconnecté !"
      redirect_to root_path
   end
end
```

Correction n°3

- Corriger l'application pour que le lien entre les articles et les utilisateurs soit implicite et que la création d'un article soit inaccessible si non connecté.

```
class ArticlesController < ApplicationController
   def new
      if session[:user].nil? then
         flash.new[:error] = "Connexion requise pour continuer !"
         redirect_to signin_path
         return
      end
      # [...]
   end
   def create
      @article = Article.new(params[:article])
      @article.user = session[:user]
      # [...]
   end
end
```

Correction n°4

- Interdire à l'utilisateur connecté de modifier un autre profil que le sien, et interdire globalement la modification si non connecté.

```
class UserController < ApplicationController
   def edit
      user = session[:user]
      if user.nil? then
         flash.new[:error] = "Connexion requise pour continuer !"
         redirect_to signin_path
         return
      end
      if user.id != params[:id] then
         flash.new[:error] = "Ce n'est pas votre profil !"
         redirect_to root_path
         return
      end
   end
end
```

Et maintenant ?

- Ça commence à devenir très dense... il faut absolument pratiquer pour que les habitudes entrent !
- Pour revoir les bases de Ruby on Rails, je vous conseille de faire les exercices de Rails for Zombies.

http://www.railsforzombies.org/

That’s all folks!

Questions ?

Je vous invite à lire la documentation sur http://guides.rubyonrails.org !

Remerciements

- Merci à Michael Hartl pour son tutorial Rails très riche http://www.railstutorial.org/chapters/
- Merci à Envy Labs pour leur *Rails for Zombies* http://www.railsforzombies.com/