- THP Starter Pack -

Indispensables terminal

- $ cd path
- $ ls -tree => voir l'arborescence
- $ls -lar => voir les fichiers cachés
- $ mkdir dossier
- $ touch fichier
- $ cp "/dossier/fichier.source" "/dossier2/fichier_newdestination" => copier un fichier
- $ mv fichier_new_name newdestination => renommer déplacer

Git Hub

Déposer son repo depuis le terminal

- $ cd path
- $ git init
- $ git status
- $ git add + nom_fichier ou git add -A
- $ git status
- $ git commit -m "initial commit"
- $ git remote add origin https://....=> mettre son lien de repo Github
- $ git remote
- $ git push origin master

Cloner un autre repo et envoyer un commit

- $ git clone https://.....
- $ cd path sur mon ordi
- $ touch machin.html => ex ajouter un doc html
- $ git commit -m "modif Larat"
- $ git status
- git push origin master

Envoyer un changement depuis son ordinateur sur son repo

- $ cd path
- $ git status
- $ git add fichier ou git add -A
- $ git status
- $ git commit -m "changement de titre"

- git status
- git push (choisir la branche)

Heroku

- $ heroku create nom_app
- $ cd mon_app
- $ bundle install
- $ git add .
- $ git commit -m “prems event_brite”
- $ git push heroku master
- $ heroku run rails db:migrate
- $ heroku ps:scale web=1
- $ heroku ps
- $ heroku open
- $ heroku run rails console => *pour faire un test sur la console heroku*

Création d’une app Rails

- $ rails _5.2.3_ new -d postgresql mon_app
- # gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
- $ ajouter tous les gems adéquats
- $ bundle install
- $ cd le_chemin_de_mon_app/
- $ rails db:create
- $ rails db:seed ou rails db:setup
- $ rails s

Commandes : correction rapide

- $ cd path
- mettre le # dans le gemfil devant tzinfo...
- $ bundle install
- $ rails db:create => pr que la data base se fasse bien (cf dans config)
- $ rails db:setup (remplace db: migrate et db:seed)
- $ rails s
- go to localhost:3000

Rappel Migrations (>db>dans les tabs de migration)

- $ rails g migration CreatePostsTable
- $rails db:migrate
- $rails db:rollback => pour revenir en arrière

- $ rails d migration VERSION=0 => pour revenir en arrière en suppr tous les seeds

Un changement dans un model implique nécessairement une migration à mettre à jour !

Ex: Rename

- $ rails g migration RenamePostTitleToName

je mettrai dans ma db > migrate > create...posts

```
def change
      change_table : post do |t|
      t.rename :title, :name
end
```

Ex: Faire une liaison

- $ rails g migration add_category_to_posts category:references

Rappel Validations>(models)

Toujours dans les models !

Les validations de base

validation simple

- `validates :name, presence: true`

validation à message

- `validates :name, presence { message : 'ne doit pas être vide'}`

validation type longueur

- `validates :name, length: { minimum : 3 }`
- `validates :name, length: { maximum : 3 }`
- `validates :name, length: { is : 3 }`
- `validates :name, length: { in : 3..20 }`

validation type format

- `validates :name, format: { with : / \A[a-zA-Z]+\z / }` *=> format non caract spec.*

validation d'unicité

- `validates :name, uniqueness: true`

validation à critères multiple dont celui d'un e-mail

- `validates :email, presence: true, uniqueness: true, format: {with: /\A[^@\s]+@([^@\s]+\.)+[^@\s]+\z/, message: "email adress please"}`

validation de type acceptance (ex pr des cases à cocher si dans un formulaire j'ai une checkboxe)

- validates :name, acceptance: true

validation pour mdp

- validates :password, confirmation: true
  password_confirmation

les validations à certaines moments seulement

ex: on voudrait ne faire une validation qu'à la création d'un contenu cf tuto Grafikart

- validates :name, length: { is: 2}, on: :create

Il y a un create => on doit aller dans le controller correspondant au model et intégrer une notion de validité. *ex dans une def dans le posts_controller.rb*

```
def create
post = Post.new(post_params)
if post.valid?
post.save
redirect_to post_path(post.id), success: "Article créé avec succès"
else
@post = post
render 'new'  => redirige vers la page de création
```

validation avec allow ex pour ne pas tout faire bugger

- validates :name, length: { is: 2}, on: :create, allow_blank: true
- validates :name, length: { is: 2}, on: :create, allow_nil: true

validation stricite => un message d'erreur est automatiquement envoyé

- validates :name, length: { is: 2}, on: :create, strict: true
- validates :name, length: { is: 2}, on: :create, strict: AEZAEZ

validation de condition avec sa propre méthode (la méthode doit aller à la fin !

- validates :name, length: { is: 2}, if :check_content_2
- validates :name, length: { is: 2}, unless :check_content_2

et en-dessous ma def

```
def check_content_2
if name_length != 2
errors.add(:name, 'le champ doit contenir 2 caractères')
end
```

ou on peut aussi écrire

```
def check_content_2
if name_length != 2
```

```
errors.add(:name, :not_2, { message :}
```

Rappel du CRUD

	Méthodes dans controllers et views	
CREATE	new	create
READ	show	index
UPDATE	edit	update
DELETE	destroy	

Rappel structure de l'app

Controllers (dans app> controllers): effectuent les actions, toujours se dire que les méthodes / actions sont dans des controllers. Le nom est tjrs en snake_case et au pluriel !

Par défaut tout controller **doit contenir toutes les méthodes du CRUD**
toutes **les routes suivent la convention REST** donc pointent vers des controllers qui établissent une action CRUD. Ainsi partons du principe que dans chaque fichier controller je vais avoir:

- **une def index :** méthode qui récupère les infos et les envoie à la view index.html.erb en affichage
- **une def show:** méthode qui récupère les infos et les envoie à la view show
- **une def new:** méthode qui crée une info et l'envoie à la view du formulaire de remplissage new.html.erb
- **une def create** : méthode qui crée l'info à partir du formulaire de remplissage new.html.erb soumis par l'utilisateur. Le contenu sera accessible dans le hash params. Elle redirige souvent vers la méthode show. (qui exécutera la view)
- **une def edit**:méthode qui récupère l'info et l'envoie à la view edit.html.erb pour affichage du formulaire de modif
- **une def update** : méthode qui met à jour l'info à partir du contenu du formulaire edit.html.erb. Le contenu sera accessible dans le hash params. Une fois les modifs faites elle redirige vers la méthode show
- **une def destroy** :méthode qui récupère l'info et la détruit en base. Une fois l'info détruite, elle redirige en méthode index.

$ rails g controller message message show
=> cette commande ne créé pas la route et la view, il faut le faire à la main ensuite

EXCEPTIONS : sauf si ces actions sont directement liées à la BDD. *Ex un email de bienvenue utilisateur dès ajout d'un utilisateur est lié à un ajout de ligne dans la BDD dans le tableau Users donc c'est plus logique que l'action (méthode) soit rattachée au Model User*

Models (dans app >models) : les tableaux => toujours bien actualiser les migrations , faire `$rails db:rollback` puis repasser la migration avec `$rails db:migrate` pour être sûr que tout fonctionne
=> les Validations sont à inclure dans les models
=> les liens de type : `has_my` ou `belongs_to` également
=> les callbacks type : `after_create`
Label toujours en Camel_Case et singulier

`$ rails g model User first_name:string last_name:string email:string`

Migrations (dans db>migrate>"create....users.") : elles doivent être le pendant des models donc toujours vérifier qu'elles soient bien à jour, surtout quand on ajoute des liens de type: `t.references :admin, index: true`

Toutes les migrations sont récapitulées dans db>schema.rb

`$ rails g migration CreatePostsTable`
`$ rails db:migrate`
`$ rails db:migrate:status`
`$ rails db:rollback`

Seeds.rb (dans db): c'est là qu'on va mettre tous les Faker et le fait de tout détruire au renouvellement pr ne pas saturer de BDD. Bien penser à faire un `$ rails db:create` pour initialiser les databases et d'ajouter la gem faker avec un `'require faker'` dans le fichier seed.

`$ rails db:seed`

ex de ce que je mets dans un fichier seed.rb

```
User.destroy_all
Event.destroy_all
Attendance.destroy_all

  10.times do |i|

                              user = User.create!(
                                  first_name: Faker::Name.first_name,
                                  last_name: Faker::Name.last_name,
                                  email: Faker::Internet.email,
                                  description: Faker::Lorem.paragraphs,
```

Les views: (dans app>views) pas de views pas de page. Bien penser à mettre .erb dans l'enregistrement.

Les routes : (config > routes.rb) obéissent à la convention REST et doivent donc pointer vers les controllers qui établissent une action CRUD.

$ rails routes permet de voir toutes les routes

Par convention on écrit :

```
resources :events
```

Resteindre les routes

```
resources  :gossips, except [ :destroy ]

resources :events, only [ :new, :create, :index, :destroy ]
```

La route de ma home

```
root 'events#index'
root to: 'pages #home'
```

Moyen mémo-technique : "JE VOIS UN CONTRÔLEUR SUR LA ROUTE"

1 controller 1 route 1 view

Les paths: correspondent aux routes créées dans config > routes.rb. On en a besoin pour les liens car on ne peut pas utiliser les <a href> sauf pour la home, on écrit plutôt <%=link_to "clique-ici", le_patch %>

Pour faire un path, il suffit sinon de prendre le nom du controller + suffixe "_path"

NB: Toutes les fichiers avec 'application' désignent des fichiers à effet parents, concernent des caractéristiques génériques

NB: <%= permet de mettre du ruby dans de l'html et de l'afficher si signe "="

NB: il faut bien déclarer ses variables dans les modèles et controllers pour les utiliser en views

les tests console rails(>terminal)

$ rails c

$ u = User.find(params[:un_nom_de_variable]

$ User.create (first_name:"Zouzou", email:"zouzou@yopmail.com")

$ u = User.find(3)

$ i = User.find_by(first_name: 'David')

$ e = User.where(first_name: 'David', is_admin: true).order(created_at: 03-02-01)

=> donne tous les David admin triés par date chronologique de création d'entrée

$ e.update(first_name:"Tété")

Rappel :

on peut ajouter toutes les méthodes:

- .all
- .new
- .save
- .update
- .destroy
- .last
- .first
- .find (X)
- .where(clé:"valeur")
- .errors => *permet d'afficher la raison de l'erreur*
- article1.user *=> renvoie l'objet associé user ex : Michel 25 ans...*
- article1.users *=> renvoie un array de users*

Les fonctionnalités spé(>models)

class_name : permet d'utiliser un nom

```
belongs_to :sender, class_name: "User"
```

optional : ex on ne veut pas de rollback qui fait tout planter si j'ai un article sans auteur

```
belongs_to :author, optional: true
```

dependent : ex pr liens docteurs patients, si un patient annule, le rdv avec le docteur est annulé

```
has_many :appointments, dependent: :destroy
```

les plus courantes avec belongs_to

- `:autosave`
- `:class_name`
- `:counter_cache`
- `:dependent`
- `:foreign_key`
- `:primary_key`
- `:inverse_of`
- `:polymorphic`
- `:touch`
- `:validate`
- `:optional`

Les associations (>models)

- `belongs_to`

- has_one
- has_many
- has_many :through
- has_one :through
- has_and_belongs_to_many

Les formats (>controllers liés à models si on veut)

cf vidéo 9/30 Grafikart pour voir en living code. ex on veut une vue qui renvoie du json si on fait .json dans l'url
On utilise la méthode respond_to, ex dans ma def index

```
respond_to do |format|
format.html
format.json do
render json: @posts (ex si on veut des articles en json)
end
format.xml { render xml: @posts }
```

On peut mettre des variantes plus précises si par ex je ne veut afficher en json que le nom, la date de création et l'id

```
format.json { render json: @posts.as_json(only: [:name, :created_at, :id]) }
```

par ex on peut lier au model ex ici Post < Application Record et y créer une méthode as_json

```
def as_json(optional = nil)
      super(only: [:name, :id, :created_at])
end
```

/!\ Dans config > Initializers, on peut enlever du commentaire le Mim tips pr paramétrer un format rtf (toujours relancer avec rails s)

validation d'url avec format exemple

- `validates :slug, format: { with /\A[a-z0-9\-]+\z/}, uniqueness: true`

=> on accepte un format d'url avec des lettres, des chiffres, des tirets, tout le reste des caractères non ! On veut que ce soit répété plusieurs fois d'où le '+'

Les callbacks(>models)

Servent à mettre en place des actions à un certain moment de la vie d'un objet

cf vidéo 11/30 Grafikart pour voir en living code
cf doc en lien

Toujours bien de les déclarer en private

```
class User < ApplicationRecord
  before_validation :normalize_name, on: :create

  # :on takes an array as well
  after_validation :set_location, on: [ :create, :update ]

  private
    def normalize_name
      self.name = name.downcase.titleize
    end

    def set_location
      self.location = LocationService.query(self)
    end
end
```

Les callbacks de création

- before_validation
- after_validation
- before_save
- around_save
- before_create
- around_create
- after_create
- after_save
- after_commit/after_rollback

les callbacks de mise à jour

- before_validation
- after_validation
- before_save
- around_save
- before_update
- around_update
- after_update
- after_save
- after_commit/after_rollback

les callbacks de destruction

- before_destroy
- around_destroy
- after_destroy
- after_commit/after_rollback

Bootstrap

Copier coller le code ci-dessous dans app >views >layout dans le head

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css
"
integrity="sha384-9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKG
j7Sk" crossorigin="anonymous">
<script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"
integrity="sha384-OgVRvuATP1z7JjHLkuOU7Xw704+h835Lr+6QL9UvYjZE3Ipu6Tp75j7Bh/kR
0JKI" crossorigin="anonymous"></script>
```

Mettre la barre de nav : aller dans la page exemples, cliquer sur l'exemple intéressant clic droit inspecter, editer as html et copier coller une div container ou une nav par ex

Tips méthodes

- parameterize pour générer des url dans un form par exemple (dans mon controller)
- gem jbuilder => permet de rendre du json avec une vue spécifiqu
- $ rails g scaffold User username:string bio:string

=> faire d'un coup model controller view migration et database

- mettre un nom en grisé dans une barre de recherche ou en formulaire qui disparaît quand on écrit par-dessus => attribut placeholder
- mettre une image qui sera prise en compte par heroku : il faut prendre une image en lien et non dans le dossier image.

Quant au code css lié qui intègre 'background-image' il doit être dans un fichier .scss et non css ex:
>app>assets>stylesheets>fichier.scss

Exemple du jumbotro avec image en css rappel

```
.jumbotron-hero {
```

```
  background-image:
url("https://images.unsplash.com/photo-1483706600674-e0c87d3fe85b?ixlib=rb-1.2.1&i
xid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=1082&q=80");
  background-position: center;
  background-size: cover;
  height: 400px;
}
```

Exemple de code à mettre dans application.scss avc des polices et un carrousel

```
@import "font-awesome-sprockets";
@import "font-awesome";

a {
	text-decoration: none !important;
}

a:hover{
	background-color: #fff0 !important;
}

.card-event:hover {
	background-color: #19a2b845;

}
.carousel-control-next:hover {
	background-color: transparent;
}
.carousel-control-prev:hover {
	background-color: transparent;
}
input.body {
  width:95%;
}
```

Les filtres (>controllers)

- before_action
- after_action
- arround_action

on met toujours le filtre en haut et la méthode en private en bas de la classe

Messages flash /success/errors (>controllers et dans >views)

ex pr le message flash à création d'un event

```
def create
    @event = Event.new(title: params[:title],
      description: params[:description],
      location: params[:location],
      price: params[:price],
      start_date: params[:start_date],
      duration: params[:duration],
      admin_id: current_user.id)

    if @event.save # essaie de sauvegarder en base @gossip
        flash[:success] = "You successfully created an event"
        redirect_to :controller => 'events', :action => 'show', id: @event.id
    else
      # This line overrides the default rendering behavior, which
      # would have been to render the "create" view.
      flash.now[:danger] = "Error with the account creation"
      render :action => 'new'
    end
  end
```

on peut mettre un flash dans redirect

```
redirect_to posts_path, flash: { success: “Article créé avec succès” }
```

par défaut la méthode raccourcie se fait avec 'notice' et 'error'

```
redirect_to posts_path, notice: “Article enregistré”
redirect_to posts_path, error: “Article enregistré”
```

méthode pr ajouter plusieurs messages, aller dans ApplicationController

créer une méthode `add_flash_types :success, :danger`

ça c'était notre code de base : `redirect_to posts_path, notice: “Article enregistré”`

maintenant on pourra mettre dans les controllers,

```
redirect_to posts_path, success: “Article enregistré”
redirect_to posts_path, danger: “Article non sauvegardé”
```

Pour faire un flash instantané qui n'apparaît pas tout le temps, ajouter la méthode `.now`

```
ex : dans PostController
def index
flash.now [:success] = “Salut”
```

Mettre les flash dans la view également

```
<% flash.each.do |name, msg|%>
<div class=”alert alert-<%= name %><% =msg %></div>
```