Introduction au développement Web avec le framework Phoenix

Novembre 2015
Mickaël Rémond - @mickael

Qu'est-ce que Phoenix ?

- Framework de développement Web
- Développement lancé en Août 2014 avec un focus sur les websockets
- Inspiration venant de différents frameworks: Ruby on Rails, Clojure Ring, Scala Playframework, …
- Version 1.0 sortie en Août 2015.

- **Élégance** grâce à la syntaxe d'Elixir et à son système de macros (DSL)
- **Performance** et clustering grâce à la VM Erlang
- **Productivité** avec des outils de base de données comme Ecto
- **Fonctionnalités:** couvre le support des **pages web dynamique** et l'interaction temps réel avec les **websockets**.

Performance

Les performances d'Elixir pour le développement Web sont parmi les meilleures aujourd'hui.

Quelques benchmarks notables:

- **Web traditionnel**: Nombres de requêtes par secondes, temps de réponse et stabilité
- **Web Temps Réel**: Connexions concurrentes avec websocket.

Performance: requêtes / seconde

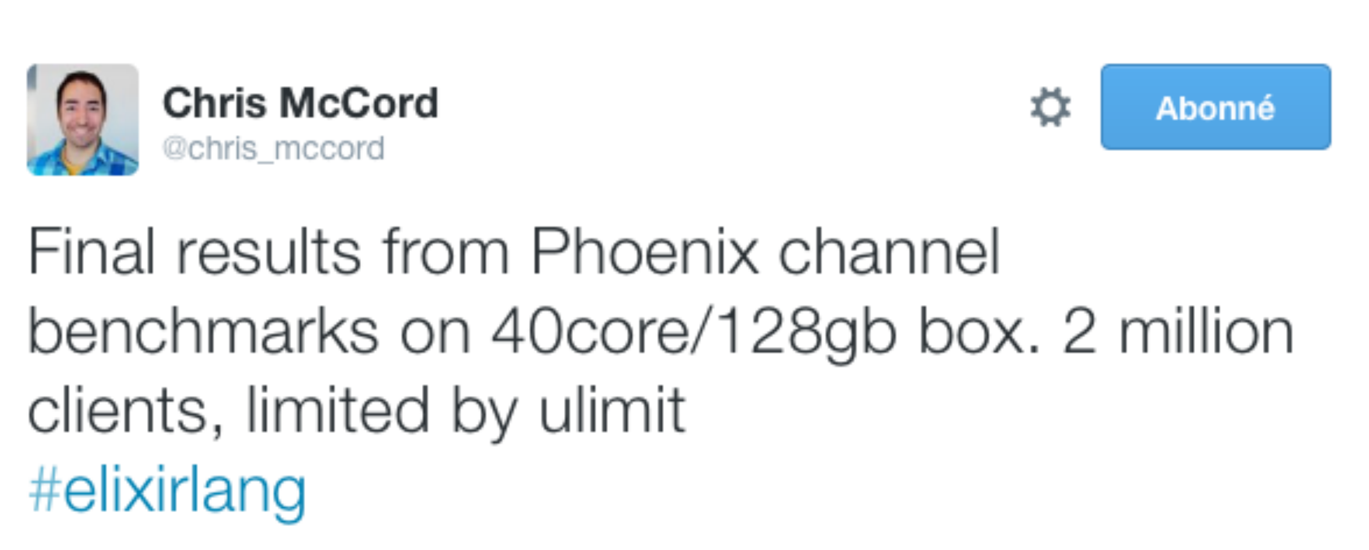
Comparative Benchmark Numbers @ Rackspace (2.5$/h)

Framework	Throughput (req/s)	Latency (ms)	Consistency (σ ms)
Plug	198328.21	0.63ms	2.22ms
Phoenix 0.13.1	179685.94	0.61ms	1.04ms
Gin	176156.41	0.65ms	0.57ms
Play	171236.03	1.89ms	14.17ms
Phoenix 0.9.0-dev	169030.24	0.59ms	0.30ms
Express Cluster	92064.94	1.24ms	1.07ms
Martini	32077.24	3.35ms	2.52ms
Sinatra	30561.95	3.50ms	2.53ms
Rails	11903.48	8.50ms	4.07ms

Performance: Connexions concurrentes

[2 millions concurrent clients receiving on a single channel]()

L'équipe de Phoenix a mesuré la performance pour le web temps réel:



Les raisons de la performance

- Le code est entièrement compilé en code Erlang.
- L'architecture est basé sur la VM Erlang, ses process légers et son scheduler.
- Le router est compilé et utilise le pattern matching d'Erlang.
- Les templates sont pré-compilés en fonctions. Ce sont des "linked lists" qui s'ajoute efficacement sans copie.
- Les languages fonctionnels permettent de matcher le flux de requêtes / réponses sans le surcoût de l'architecture objet.

Concurrence "native"

Pas besoin de "PhoenixDelayedJob" ou "ElixirResque" !

```
company_task  = Task.async(fn -> find_company(cid) end)
user_task     = Task.async(fn -> find_user(uid) end)
cart_task     = Task.async(fn -> find_cart(cart_id) end)

company = Task.await(company_task)
user = Task.await(user_task)
cart = Task.await(cart_task)
```

Les concepts

- **Connection:** La connection est la structure de base qui est utilisée par les composants d'architecture de Phoenix.
- **Endpoint:** Il s'agit de la chaine de traitement définie pour votre application. Il s'agit d'un enchainement de fonctions. En général vous avez un endpoint mais vous pouvez en avoir un autre pour le backend d'admin par exemple.
- **Router:** Le router est plus fin et divise votre application en plusieurs pipelines qui sont aussi des enchainements de fonctions.
- **Controller:** Le controller permet comme d'habitude de controller les opérations appliquées sur une URL donnée.
- **View:** Les vues controllent le rendu des pages, ou des appels API.

Le flux de traitement d'une requête est le suivant:

```
request |> connection
    |> endpoint
    |> router
    |> pipeline
    |> controller
    |> view
```

Les *plugs*

Les ***plugs*** sont les maillons de la chaîne de traitement des requêtes:

- Ce sont des fonctions
- Ils acceptent une ***connection*** en entrée et renvoie une ***connection***.
- Ils se composent entre eux pour faire des plugs plus complexes
    - Exemple: pipeline

De cette manière les plugs définissent une manière standard de composer le flux des appels de fonctions dans le framework Phoenix.

Les ***plugs*** sont utilisés pour:

- Les étapes du router
- les actions des controllers
- La définition des endpoints
- L'authentification
- Les sessions
- les parsing du json
- le logging
- les contenus statics
- …

Les modèles

Dans le modèle MVC, le contenu lui même est fourni par le modèle.

- **Model:** Le modèle peuvent gérer des objets en mémoire, mais la plupart du temps, ils sont en base de données.
- **Repos:** Les ***Repos*** définissent un ou plusieurs backend de données pour une application.
- **Ecto:** Framework de persistence.

La démo: Gastronokids

Application web permettant aux enfants de laisser un avis sur un restaurant.

La démo sera simplifiée en reposant sur:

- Un model "post", qui contient aussi le nom est l'adresse du restaurant.
- L'usage du ***scaffolding*** pour accélérer la génération du HTML.
- La base locale SQLite.

L'environnement

Les prérequis:

- Erlang et Elixir
- Node pour la compilation des assets

Installation de Hex (gestionnaire de dépendances):

```
$ mix local.hex
```

Installation de Phoenix:

```
$ mix archive.install
https://github.com/phoenixframework/phoenix/releases/download/v1.0.3/phoenix_new-
1.0.3.ez
```

Création de la structure de l'application

Création de la structure de l'application en utilisant SQLite:

```
$ mix phoenix.new gastronokids --database sqlite
* creating gastronokids/config/config.exs
...
Fetch and install dependencies? [Yn] Y
* running npm install && node node_modules/brunch/bin/brunch build
* running mix deps.get
```

La base de données par défaut est Postgres.

Exploration de la structure de l'application

Le répertoire du projet a été créé:

```
$ cd gastronokids
```

La commande suivante compile le projet et crée la base de données:

```
$ mix ecto.create
```

La base de données est vide pour le moment:

```
$ file db/gastronokids_dev.sqlite
db/gastronokids_dev.sqlite: empty
```

Pour démarrer l'application:

```
$ mix phoenix.server
```

ou:

```
$ iex -S mix phoenix.server
```

Le serveur tourne alors sur: http://localhost:4000

Le fichier mix.exs

`mix. exs`
définit l'application et ses dépendances:

```
$ cat mix.exs

defmodule Gastronokids.Mixfile do
  use Mix.Project

  def project do
    [app: :gastronokids,
     version: "0.0.1",
     elixir: "~> 1.0",
     elixirc_paths: elixirc_paths(Mix.env),
     compilers: [:phoenix] ++ Mix.compilers,
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     aliases: aliases,
     deps: deps]
  end

  # Configuration for the OTP application.
  #
  # Type `mix help compile.app` for more information.
  def application do
    [mod: {Gastronokids, []},
     applications: [:phoenix, :phoenix_html, :cowboy, :logger,
                    :phoenix_ecto, :sqlite_ecto]]
  end

  # Specifies which paths to compile per environment.
  defp elixirc_paths(:test), do: ["lib", "web", "test/support"]
  defp elixirc_paths(_),     do: ["lib", "web"]

  # Specifies your project dependencies.
  #
  # Type `mix help deps` for examples and options.
  defp deps do
    [{:phoenix, "~> 1.0.3"},
     {:phoenix_ecto, "~> 1.1"},
     {:sqlite_ecto, ">= 0.0.0"},
     {:phoenix_html, "~> 2.1"},
     {:phoenix_live_reload, "~> 1.0", only: :dev},
     {:cowboy, "~> 1.0"}]
  end

  # Aliases are shortcut or tasks specific to the current project.
  # For example, to create, migrate and run the seeds file at once:
  #
  #     $ mix ecto.setup
  #
  # See the documentation for `Mix` for more info on aliases.
  defp aliases do
    ["ecto.setup": ["ecto.create", "ecto.migrate", "run
priv/repo/seeds.exs"],
     "ecto.reset": ["ecto.drop", "ecto.setup"]]
  end
end
```

Le fichier route.ex

`route.ex`
défini la structure des URL de l'application:

```
$ cat web/route.ex

defmodule Gastronokids.Router do
  use Gastronokids.Web, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", Gastronokids do
    pipe_through :browser # Use the default browser stack

    get "/", PageController, :index
    resources "/posts", PostController
  end

  # Other scopes may use custom stacks.
  # scope "/api", Gastronokids do
  #   pipe_through :api
  # end
end
```

Phoenix et le scaffolding

Phoenix propose un ensemble de générateur de code permettant de démarrer la structure de son application plus rapidement:

- phoenix.gen.html
- phoenix.gen.model
- phoenix.gen.json
- phoenix.gen.channel

Le modèle *post*

Nous allons utiliser un mécanisme comme le scaffolding pour nous aider à développer plus vite et générer certains des fichiers dont nous avons besoin.

Voici les champs simples que nous allons gérer au début:

- **Restaurant name:** name: string
- **Restaurant address:** address: string
- **User Comment:** body: string

```
$ mix phoenix.gen.html Post posts name address body:string
* creating web/controllers/post_controller.ex
* creating web/templates/post/edit.html.eex
* creating web/templates/post/form.html.eex
* creating web/templates/post/index.html.eex
* creating web/templates/post/new.html.eex
* creating web/templates/post/show.html.eex
* creating web/views/post_view.ex
* creating test/controllers/post_controller_test.exs
* creating priv/repo/migrations/20151125104650_create_post.exs
* creating web/models/post.ex
* creating test/models/post_test.exs
Add the resource to your browser scope in web/router.ex:
    resources "/posts", PostController
Remember to update your repository by running migrations:
    $ mix ecto.migrate
```

Notes:

- Le nom Post est le nom du module. posts est utilisé pour les noms de resources et schéma.
- Le type string est le type par défaut.

Le modèle généré

```
$ cat web/models/post.ex

defmodule Gastronokids.Post do
  use Gastronokids.Web, :model

  schema "posts" do
    field :name, :string
    field :address, :string
    field :body, :string

    timestamps
  end

  @required_fields ~w(name address body)
  @optional_fields ~w()

  @doc """
  Creates a changeset based on the `model` and `params`.

  If no params are provided, an invalid changeset is returned
  with no validation performed.
  """
  def changeset(model, params \\ :empty) do
    model
    |> cast(params, @required_fields, @optional_fields)
    |> validate_length(:name, min: 3, max: 30)
  end
end
```

Le fichier de migration

Le fichier de migration permet de créer ou de modifier les tables de base de données pour coller à votre modèle.

```
$ cat priv/repo/migrations/20151125104650_create_post.exs

defmodule Gastronokids.Repo.Migrations.CreatePost do
  use Ecto.Migration

  def change do
    create table(:posts) do
      add :name, :string
      add :address, :string
      add :body, :string

      timestamps
    end
  end
end
```

Notes:

- **timestamps** est une valeur spéciale pour tracer les dates de création et de modification.
- Ecto ajoute un champs **id** contenant une clé primaire s'il n'y en a pas.
- La gestion des migrations avec le code permet de s'assurer du bon versionning du schéma de base de données.
- Le language d'Ecto permet d'exprimer aussi bien l'upgrade du schéma de base de données que le **rollback**. C'est possible car nous Ecto génère lui même le SQL.

Le controleur Post

```
$ cat web/controllers/post_controller.ex

defmodule Gastronokids.PostController do
  use Gastronokids.Web, :controller

  alias Gastronokids.Post

  plug :scrub_params, "post" when action in [:create, :update]

  def index(conn, _params) do
    posts = Repo.all(Post)
    render(conn, "index.html", posts: posts)
  end

  def new(conn, _params) do
    changeset = Post.changeset(%Post{})
    render(conn, "new.html", changeset: changeset)
  end

  def create(conn, %{"post" => post_params}) do
    changeset = Post.changeset(%Post{}, post_params)

    case Repo.insert(changeset) do
      {:ok, _post} ->
        conn
        |> put_flash(:info, "Post created successfully.")
        |> redirect(to: post_path(conn, :index))
      {:error, changeset} ->
        render(conn, "new.html", changeset: changeset)
    end
  end

  def show(conn, %{"id" => id}) do
    post = Repo.get!(Post, id)
    render(conn, "show.html", post: post)
  end

  def edit(conn, %{"id" => id}) do
    post = Repo.get!(Post, id)
    changeset = Post.changeset(post)
    render(conn, "edit.html", post: post, changeset: changeset)
  end

  def update(conn, %{"id" => id, "post" => post_params}) do
    post = Repo.get!(Post, id)
    changeset = Post.changeset(post, post_params)

    case Repo.update(changeset) do
      {:ok, post} ->
        conn
        |> put_flash(:info, "Post updated successfully.")
        |> redirect(to: post_path(conn, :show, post))
      {:error, changeset} ->
        render(conn, "edit.html", post: post, changeset: changeset)
    end
  end

  def delete(conn, %{"id" => id}) do
    post = Repo.get!(Post, id)

    # Here we use delete! (with a bang) because we expect
    # it to always work (and if it does not, it will raise).
    Repo.delete!(post)

    conn
    |> put_flash(:info, "Post deleted successfully.")
    |> redirect(to: post_path(conn, :index))
  end
end
```

La vue et les templates

La vue sert à fournir des fonctions helpers, utilisable dans les templates. Ici, elle est minimaliste:

```
$ cat web/views/post_view.ex

defmodule Gastronokids.PostView do
  use Gastronokids.Web, :view
end
```

Nous avons ensuite un template par action sur la vue, plus un formulaire d'édition. Par exemple:

```
$ cat web/templates/post/form.html.eex

<%= form_for @changeset, @action, fn f -> %>
  <%= if @changeset.action do %>
    <div class="alert alert-danger">
      <p>Oops, something went wrong! Please check the errors below:</p>
      <ul>
        <%= for {attr, message} <- f.errors do %>
          <li><%= humanize(attr) %> <%= message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="form-group">
    <%= label f, :name, "Name", class: "control-label" %>
    <%= text_input f, :name, class: "form-control" %>
  </div>

  <div class="form-group">
    <%= label f, :address, "Address", class: "control-label" %>
    <%= text_input f, :address, class: "form-control" %>
  </div>

  <div class="form-group">
    <%= label f, :body, "Body", class: "control-label" %>
    <%= text_input f, :body, class: "form-control" %>
  </div>

  <div class="form-group">
    <%= submit "Submit", class: "btn btn-primary" %>
  </div>
<% end %>
```

Changement de route

L'outil de génération nous laisse modifier notre route:

```
$ vim web/route.ex

...
  scope "/", Gastronokids do
    pipe_through :browser # Use the default browser stack

    get "/", PageController, :index
    resources "/posts", PostController
  end
...
```

resource permet d'ajouter toutes les actions ReST d'un coup, au lieu d'ajouter une ligne pour get, post, delete, etc.

Lorsque la route est ajoutée, il est possible de compiler le code et de générer la migration.

Exécuter la migration

Attention, si vous essayer de migrer avec d'ajouter la route, la compilation du controller échoue avec:

```
function post_path/2 undefined
```

```
$ mix ecto.migrate

Compiled web/models/post.ex
Compiled web/views/page_view.ex
Compiled web/views/layout_view.ex
Compiled web/views/error_view.ex
Compiled web/router.ex
Compiled web/controllers/page_controller.ex
Compiled web/controllers/post_controller.ex
Compiled lib/gastronokids/endpoint.ex
Compiled web/views/post_view.ex
Generated gastronokids app

12:02:16.121 [info]  == Running
Gastronokids.Repo.Migrations.CreatePost.change/0 forward

12:02:16.122 [info]  create table posts

12:02:16.126 [info]  == Migrated in 0.0s
```

Le schéma de la base de données a bien été mis à jour:

```
$ sqlite3 db/gastronokids_dev.sqlite .schema

CREATE TABLE "schema_migrations" ("version" BIGINT PRIMARY KEY, "inserted_at"
DATETIME);
CREATE TABLE "posts" ("id" INTEGER PRIMARY KEY AUTOINCREMENT, "name" TEXT,
"address" TEXT, "body" TEXT, "inserted_at" DATETIME NOT NULL, "updated_at"
DATETIME NOT NULL);
```

Lancer l'application

Il est maintenant possible de lancer le serveur et de créer des posts:

```
$ iex -S mix phoenix.server
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:4:4] [async-threads:10]
[hipe] [kernel-poll:false] [dtrace]

[info] Running Gastronokids.Endpoint with Cowboy on http://localhost:4000
Interactive Elixir (1.1.1) - press Ctrl+C to exit (type h() ENTER for help)
25 Nov 17:07:22 - info: compiled 5 files into 2 files, copied 3 in 1923ms
iex(1)>
```

Le server est disponible sur: http://localhost:4000/posts

Il est possible de créer, supprimer, modifier des posts.

Le scaffolding constitue une bonne base pour créer une structure de projet rapidement ou des écrans d'admin.

Get Started

Listing posts

Name	Address	Body

New post

Get Started

New post

Oops, something went wrong! Please check the errors below:

- Name can't be blank
- Address can't be blank
- Body can't be blank

Name

Address

Body

Submit

Back

Get Started

Post created successfully.

Listing posts

Name	Address	Body	
Le Balbuzard Café	54, rue René Boulanger 75010 Paris	Cuisine corse savoureuse	Show Edit Delete

New post

Utilisation du shell Elixir

```
$ iex -S mix
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:4:4] [async-threads:10]
[hipe] [kernel-poll:false] [dtrace]

Interactive Elixir (1.1.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> alias Gastronokids.Repo
nil
iex(2)> alias Gastronokids.Post
nil
```

```
iex(3)> Repo.insert(%Post{name: "La cantine de Zoé", address: "rue du
Faubourg Poissonière", body: "Cantine de quartier, pour un repas de midi"})
[debug] BEGIN [] OK query=3.5ms queue=27.5ms
[debug] INSERT INTO "posts" ("address", "body", "inserted_at", "name",
"updated_at") VALUES (?1, ?2, ?3, ?4, ?5) ;--RETURNING ON INSERT "posts","id"
["rue du Faubourg Poissonière", "Cantine de quartier, pour un repas de midi",
{{2015, 11, 25}, {16, 15, 43, 0}}, "La cantine de Zoé", {{2015, 11, 25}, {16,
15, 43, 0}}] OK query=12.2ms
[debug] COMMIT [] OK query=1.3ms
{:ok,
 %Gastronokids.Post{__meta__: #Ecto.Schema.Metadata<:loaded>,
  address: "rue du Faubourg Poissonière",
  body: "Cantine de quartier, pour un repas de midi", id: 2,
  inserted_at: #Ecto.DateTime<2015-11-25T16:15:43Z>, name: "La cantine de
Zoé",
  updated_at: #Ecto.DateTime<2015-11-25T16:15:43Z>}}
```

```
iex(7)> Repo.all(Post)
[debug] SELECT p0."id", p0."name", p0."address", p0."body", p0."inserted_at",
p0."updated_at" FROM "posts" AS p0 [] OK query=5.4ms
[%Gastronokids.Post{__meta__: #Ecto.Schema.Metadata<:loaded>,
  address: "54, rue René Boulanger 75010 Paris",
  body: "Cuisine corse savoureuse", id: 1,
  inserted_at: #Ecto.DateTime<2015-11-25T11:07:15Z>, name: "Le Balbuzard
Café",
  updated_at: #Ecto.DateTime<2015-11-25T11:07:15Z>},
 %Gastronokids.Post{__meta__: #Ecto.Schema.Metadata<:loaded>,
  address: "rue du Faubourg Poissonière",
  body: "Cantine de quartier, pour un repas de midi", id: 2,
  inserted_at: #Ecto.DateTime<2015-11-25T16:15:43Z>, name: "La cantine de
Zoé",
  updated_at: #Ecto.DateTime<2015-11-25T16:15:43Z>}]
```

```
iex(6)> import Ecto.Query
nil
iex(7)> Repo.one(from p in Post, where: p.name == "La cantine de Zoé",
select: p.address)
[debug] SELECT p0."address" FROM "posts" AS p0 WHERE (p0."name" = 'La cantine
de Zoé') [] OK query=2.0ms queue=27.7ms
"rue du Faubourg Poissonière"
```

```
iex(9)> post_count = from p in Post, select: count("*")
#Ecto.Query<from p in Gastronokids.Post, select: count("*")>
iex(10)> Repo.one from p in post_count, limit: 1
[debug] SELECT count ('*') FROM "posts" AS p0 LIMIT 1 [] OK query=0.3ms
2
```

Créer son propre plug

La possibilité pour de créer des plugs pour son app permet de gérer certains contrôles très simplement.

Par exemple pour l'authentification:

```
$ cat web/controllers/auth.ex

defmodule Gastronokids.Auth do
  import Plug.Conn

  def init(opts) do
    Keyword.fetch!(opts, :repo)
  end

  def call(conn, repo) do
    user_id = get_session(conn, :user_id)
    user    = user_id && repo.get(Gastronokids.User, user_id)
    assign(conn, :current_user, user)
  end

  def login(conn, user) do
    conn
    |> assign(:current_user, user)
    |> put_session(:user_id, user.id)
    |> configure_session(renew: true)
  end
end
```

Le user_id est ajouté dans la gestion des sessions de Phoenix lors du login (fonction login).

La structure `conn` contient des champs d'information sur la requête: headers, cookie, etc.

Le plug peut ensuite être utilisé dans le pipeline du navigateur pour vérifier systématiquement qu'une session et créé et que l'utilisateur existe

```
$ cat web/route.ex
...
  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
    plug Gastronokids.Auth, repo: Gastronokids.Repo
  end
...
```

Dans notre contrôleur pour les pages nécessitant une authentification, nous pouvons ajouter une fonction authenticate:

```
...
  defp authenticate(conn) do
    if conn.assigns.current_user do
      conn
    else
      conn
      |> put_flash(:error, "You must be logged in to access that page")
      |> redirect(to: page_path(conn, :index))
      |> halt()
    end
  end
...
```

Il est ensuite possible de vérifier systématiquement que l'utilisateur est authentifié en rajouter un plug dans le même contrôleur:

```
plug :authenticate when action in [:delete]
```

Les channels

Pour notre exemple, imaginons que nous voulons permettre de chatter entre les personnes intéressé par les restaurants.

Phoenix permet d'ajouter cela grâce aux websockets.

On utilise le scaffolding pour générer un module channel par défault:

```
$ mix phoenix.gen.channel Room rooms
* creating web/channels/room_channel.ex
* creating test/channels/room_channel_test.exs

Add the channel to your `web/channels/user_socket.ex` handler, for example:

    channel "rooms:lobby", Gastronokids.RoomChannel
```

Activation de la route pour les channels

Dans le fichier **web/channels/user_socket.ex** on décommente la définition des channels de room:

```
## Channels
channel "rooms:*", Gastronokids.RoomChannel
```

Modification du template

Dans le template HTML **web/templates/layout/app.html.eex**, ajoutons des champs pour le chat et le support de Jquery:

```
...
      <%= @inner %>

    <div class="chat">
     <div id="messages"></div>
     <input id="chat-input" type="text"></input>
    </div>

    </div> <!-- /container -->
    <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="<%= static_path(@conn, "/js/app.js") %>"></script>
  </body>
</html>
```

socket.js

Editer le fichier **web/static/js/socket.js** pour avoir le bon nom de room.

On en profite pour ajouter le code pour l'envoie et l'affichage de messages:

```
...
socket.connect()

let channel           = socket.channel("rooms:lobby", {})
let chatInput         = $("#chat-input")
let messagesContainer = $("#messages")

chatInput.on("keypress", event => {
  if(event.keyCode === 13){
    channel.push("new_msg", {body: chatInput.val()})
    chatInput.val("")
  }
})

channel.on("new_msg", payload => {
  messagesContainer.append(`<br/>[${Date()}] ${payload.body}`)
})

channel.join()
  .receive("ok", resp => { console.log("Joined successfully", resp) })
  .receive("error", resp => { console.log("Unable to join", resp) })

export default socket
```

Activer le support des sockets Phoenix dans l'application

Ensuite, pour activer le support de sockets, il faut décommenter la ligne suivante dans le fichier **web/static/js/app.js**:

```
import socket from "./socket"
```

Notre channel controller

Pour finir, il faut s'assurer que l'on broadcast correctement les événements de type **new_msg** dans notre fichier channel

```
$ cat web/channels/room_channel.ex

defmodule Gastronokids.RoomChannel do
  use Gastronokids.Web, :channel

  def join("rooms:lobby", payload, socket) do
    if authorized?(payload) do
      {:ok, socket}
    else
      {:error, %{reason: "unauthorized"}}
    end
  end

  # Channels can be used in a request/response fashion
  # by sending replies to requests from the client
  def handle_in("ping", payload, socket) do
    {:reply, {:ok, payload}, socket}
  end


  def handle_in("new_msg", %{"body" => body}, socket) do
    broadcast! socket, "new_msg", %{body: body}
    {:noreply, socket}
  end

  def handle_out("new_msg", payload, socket) do
    push socket, "new_msg", payload
    {:noreply, socket}
  end

  # Add authorization logic here as required.
  defp authorized?(_payload) do
    true
  end
end
```

Get Started

Show post

- **Name:** Le Balbuzard Café
- **Address:** 54, rue René Boulanger 75010 Paris
- **Body:** Cuisine corse savoureuse

Edit Back

[Wed Nov 25 2015 17:39:16 GMT+0100 (CET)] Salut
[Wed Nov 25 2015 17:39:26 GMT+0100 (CET)] Cela semble fonctionner

Prochaines étapes

Le projet est disponible sur Github: [ElixirParis/gastronokids]

Les contributions sont les bienvenus.

Si le projet atteint un stade exploitable, cela pourra être un projet communautaire vitrine de Paris.ex 😃

A vous de hacker !

Quelques idées:

1. Passer les pages de post à la racine en modifiant le fichier route.
2. Gestion des utilisateurs réels
3. Passage sur Postgres pour la production
4. Support des adresses emails des parents.
5. Critique avec des notes sur certains critères (attention portée aux enfants, plats pour les enfants, espace de jeu).
6. Structure de base de données avec des modèles dédiés pour les restaurants.
7. Intégration de Google Maps
8. Possibilité d'uploader des photos
9. ...