

# Elixir – Rapport pratique

## Attentes du langage choisi

L'idée de ce projet était de mettre en pratique certains des avantages principaux d'Elixir :

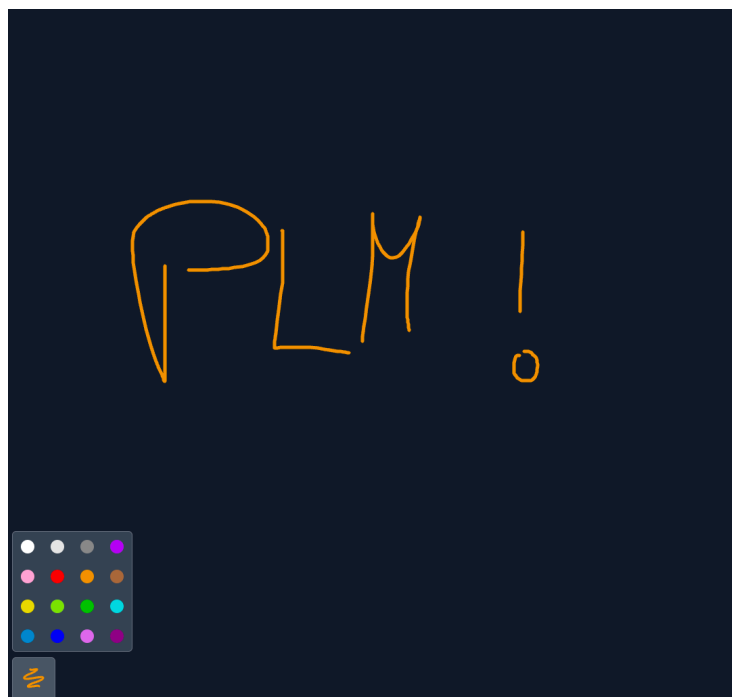
1. La concurrence
2. Son écosystème autour du web, notamment grâce au framework [Phoenix](#), spécialisé dans le temps réel et les sockets.
3. Sa faible latence, ce qui est très bénéfique dans le cas d'un serveur web
4. Sa facilité de prise en main

Un retour de ces attentes sera effectué dans le dernier chapitre de ce rapport nommé bilan.

## Réalisation du projet

### Contexte

Pour vérifier au mieux les attentes du langage, une application web interactive entre les utilisateurs a été créée. Celle-ci consiste en une application de dessin collaborative. Chaque utilisateur peut choisir une couleur et dessiner des traits sur un canvas partagé. Les autres utilisateurs reçoivent en temps réel les points des traits de tous les utilisateurs. Au chargement de la page, l'état stocké du côté du serveur est affiché à l'utilisateur. Cette application convient bien car elle demande d'envoyer un nombre conséquent d'informations en temps réel, ce qui sera utile dans le cadre des tests de montée en charge.



L'application est déployée et disponibles à tous à l'adresse suivante : [pmw.fly.dev](https://pmw.fly.dev)

(Le premier chargement peut prendre plus de temps car l'hébergement étant gratuit, il se met en veille en cas d'inactivité prolongée).

## Implémentation

Cette section va couvrir les différents points intéressants rencontrés lors du développement du projet.

### Architecture globale

Le projet suit l'architecture du framework Phoenix. Il s'agit d'un MVC classique (modèle, vue, contrôleur). Les pages sont rendues côté serveur et le navigateur reçoit directement l'HTML, contrairement à de nombreux sites actuels qui utilisent des frameworks JavaScript comme React ou autre.

Pour simplifier le projet, aucune vraie base de données n'a été utilisée mais l'état du canvas est stocké directement en mémoire. La toile se réinitialise donc si la machine redémarre mais cela a été jugé suffisant dans le cadre de ce projet.

Concernant la communication en temps réel, elle est réalisée grâce à des WebSockets. Phoenix vient nativement avec une implémentation clés en mains basée sur le mécanisme de publish-subscribe (pub-sub). Il est possible de créer un channel et de s'abonner aux messages envoyés à celui-ci. Un channel « canvas » a donc été créé pour partager les points des utilisateurs.

### Backend

Les couleurs disponibles sont stockées du côté du backend et envoyées au moteur de templating HEx sous la forme d'une variable afin d'être affichées à l'utilisateur. Cela permet notamment une plus grande évolutivité que de simplement les avoir en dur directement dans la vue.

```
defmodule PmwWeb.PageController do
  use PmwWeb, :controller

  @colors [
    '#FFFFFF',
    '#E4E4E4',
    # ...
  ]

  def home(conn, _params) do
    render(conn, :home, layout: false, colors: @colors)
  end
end
```

```
<div class="fixed bottom-0 left-0 ml-3 mb-3 z-50">
  <div id="colors" class="hidden">
    <%= for color <- @colors do %>
      <button
        data-color={color}
        class="color"
      >
```

```
<div style={"background-color: #{color}"}></div>
</button>
<% end %>
</div>
<button id="choose-color"><svg color="#FFFFFF"></svg></button>
</div>
<div id="container"></div>
```

Afin de s'abonner aux événements, le client rejoint le channel Websocket **canvas:points**. Il reçoit l'état actuel du canvas (le stockage de celui-ci sera discuté par après). Par la suite, le client a la possibilité d'émettre deux événements différents :

1. `first_point` : premier point d'une nouvelle ligne
2. `new_point` : ajout d'un point à une ligne existante

Dans les deux cas le point est stocké dans le backend et la modification est envoyée à tous les clients.

```
defmodule PmwWeb.CanvasChannel do
  use PmwWeb, :channel

  def join("canvas:points", _payload, socket) do
    state = Canvas.getAll()
    {:ok, state, socket}
  end

  def handle_in("first_point", payload, socket) do
    broadcast_from(socket, "first_point", payload)
    Canvas.add_initial(payload["id"], payload["x"], payload["y"],
payload["color"])
    {:reply, {:ok, %{}}, socket}
  end

  def handle_in("new_point", payload, socket) do
    broadcast_from(socket, "new_point", payload)
    Canvas.add(payload["id"], payload["x"], payload["y"])
    {:reply, {:ok, %{}}, socket}
  end
end
```

Pour revenir sur le stockage des différentes lignes (constituées de points), un [Agent Elixir](#) est utilisé. Cela permet de partager un state entre les différents process qui exécutent le code Elixir sans avoir de problème de concurrence. Les lignes sont stockées dans un tableau d'entiers représentant les coordonnées de chaque point : `[x1, y1, x2, y2, ...]` afin de minimiser l'espace utilisé. Pour différencier les lignes, un id unique est généré lorsque le premier point de ladite ligne est créé. Les identifiants ont un format `dateActuelle-uuid` afin d'être triés du plus ancien au plus récents lorsque l'on récupère l'entièreté des lignes.

```
defmodule Canvas do
  use Agent

  def start_link(_) do
    Agent.start_link(fn -> %{} end, name: __MODULE__)
  end

  def add_initial(line_id, x, y, color) do
    Agent.update(__MODULE__, fn state ->
      state
      |> Map.put(line_id, [x, y])
      |> Map.put("#{line_id}-color", color)
    end)
  end

  def add(line_id, x, y) do
    Agent.update(__MODULE__, fn state ->
      Map.put(state, line_id, [x, y | Map.get(state, line_id)])
    end)
  end

  def get_all() do
    Agent.get(__MODULE__, fn state ->
      state
    end)
  end
end
```

### Frontend

La partie à noter dans le code JavaScript du frontend est l'utilisation du socket directement à partir de la librairie Phoenix. En effet, il s'agit d'une implémentation Websockets plus abstraite qui permet de rejoindre facilement le channel et s'abonner aux événements créés précédemment dans le backend.

```
import { Socket } from "phoenix";

const socket = new Socket("/socket", {
  params: { token: window.userToken },
});
socket.connect();

const channel = socket.channel("canvas:points", {});
channel.join().receive("ok", (state) => {
  // Affiche le state actuel au chargement de la page
});
```

```
channel.on("first_point", (payload) => { // Crée la ligne });  
  
channel.on("new_point", (payload) => { // Ajoute le point à la ligne });
```

Toute la partie d’affichage des lignes et du canvas a été développée à l’aide de la librairie [Konva.js](#) mais je ne vais pas entrer dans les détails de celle-ci car ce rapport concerne avant tout l’utilisation d’Elixir.

### Déploiement

Afin de pouvoir tester la montée en charge de l’application, celle-ci est déployée sur [fly.io](#). Il s’agit d’une Platform as a service (PaaS) dans le genre d’Heroku. L’avantage de fly.io est qu’il propose encore un tier gratuit avec les ressources suivantes : 1 cpu partagé et 256MB de RAM. Pour rappel, l’application est accessible à l’adresse suivante : [pmw.fly.dev](#).

Pour pouvoir avoir une référence afin de comparer les performances d’Elixir, une version en Node.js de l’application a également été déployée ([pmw-node-front.fly.dev](#)). Il s’agit une version moins complète de l’application réalisée comme [tutoriel](#) par l’équipe Pimp My Wall de Baleinev afin d’initier les nouveaux collaborateurs. Malheureusement, cette application est divisée en deux : un frontend React et un backend Node. Elle a donc dû être déployée sur deux machines fly.io ce qui fausse un peu les futurs tests de montées en charge (l’application a plus de ressources allouées).

### Tests de montée en charge

Les tests de montée en charge sont réalisés grâce à l’utilitaire [Artillery](#). Il s’agit d’un CLI permettant de décrire des tests en format YAML et proposant la possibilité d’exporter les résultats sous la forme d’une page web pour visualiser facilement les graphiques. Différents types de tests sont possibles :

1. De simples requêtes HTTP
2. Des connexions et envois de données via Websocket
3. Des tests « end-to-end » en simulant un navigateur headless (sans interface)

Le premier type n’est pas très intéressant dans notre cas, en effet, de simples requêtes HTTP ne permettraient pas de tester la limitation principale de l’application qui est le transfert de données conséquentes lorsqu’un utilisateur dessine sur la toile.

Le second type, utiliser des Websocket, semble idéal. En effet, envoyer des données pseudo-aléatoires au bon endpoint websocket permettrait de simuler le dessin des lignes par un utilisateur. Malheureusement, cette solution n’a pas pu être utilisée car, comme expliqué précédemment, Phoenix utilise une abstraction au-dessus des Websockets et je n’ai pas été en mesure de faire fonctionner les tests. Ce point négatif sera plus précisé dans le dernier chapitre concernant le bilan.

Le dernier type intéressant reste donc de simuler des navigateurs pour dessiner comme de vrais utilisateurs. Le test réalisé est le suivant :

```
config:
  target: https://pmw.fly.dev
  phases:
    - duration: 60
      arrivalRate: 2
      maxVusers: 10
      name: 10 Users drawing
  engines:
    playwright: {}
  processor: "./draw.js"
scenarios:
  - engine: playwright
    flowFunction: "draw"
    flow: []
```

Le test dure 60 secondes et 2 faux utilisateurs arrivent chaque seconde pour un total de maximum 10 utilisateurs simultanés. Chaque utilisateur virtuel va exécuter le fichier **draw.js** qui est le suivant :

```
async function pickColor(page) {
  await page.locator("#choose-color").click();
  const randomColor = random(16) + 1;
  await page.locator(`button.color:nth-child(${randomColor})`).click();
}

async function line(page) {
  await page.mouse.move(randomX(), randomY());
  await page.mouse.down();

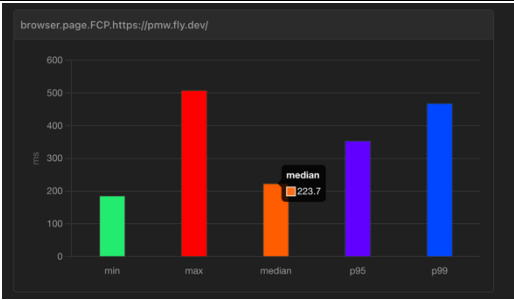
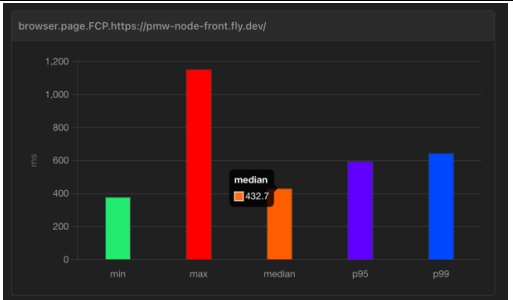
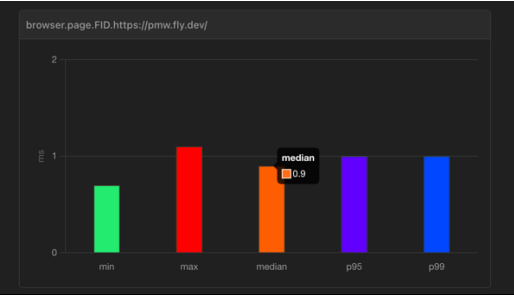
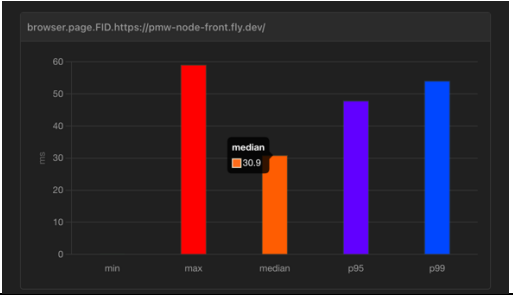
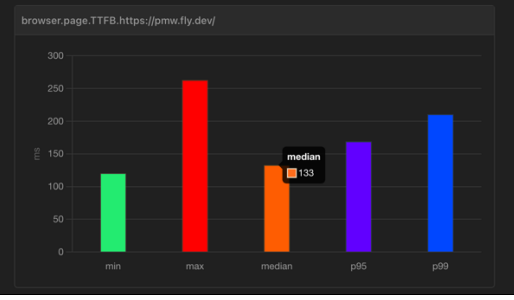
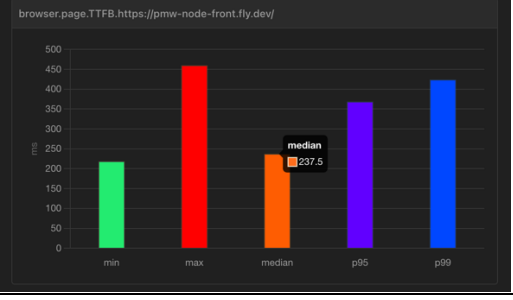
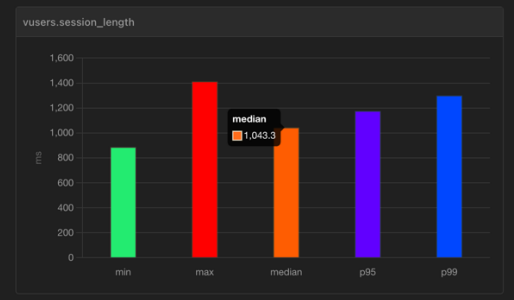
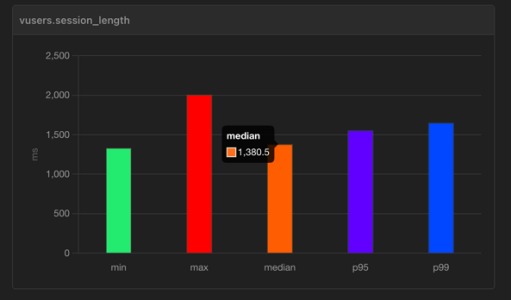
  await page.mouse.move(randomX(), randomY());
  await page.mouse.up();
}

async function draw(page) {
  await page.goto("https://pmw.fly.dev/");

  for (let i = 0; i < NB_LINES; i++) {
    await pickColor(page);
    await line(page);
  }
}
```

Celui s'occupe de dessiner 10 lignes de manière tout à fait aléatoire sur la page. Une vidéo de démonstration du résultat du test [est disponible ici](#). L'application reste fluide malgré la charge conséquente.

Cette technique fonctionne bien, le seul problème se trouve être la limitation de l'ordinateur qui exécute le test de montée en charge. En effet, bien que très récent et performant, mon ordinateur atteignait ses limites avant que les applications testées soient, elles, vraiment en difficulté. Les 10 cœurs de mon processeur arrivaient à 100% d'utilisation. C'est pour cela que le test effectué s'est contenté de 10 utilisateurs simultanés au maximum pour un total de 120 en une minute. Le tableau qui suit compare donc différentes valeurs entre l'application Elixir et la version Node afin de pouvoir quand même tirer une plus-value de ce test de montée en charge.

Application Elixir	Application React + Node																								
<b>FCP (First Contentful Paint) : délai avant l'affichage du premier élément de la page</b>																									
 <table border="1"><caption>FCP (Elixir)</caption><thead><tr><th>Metric</th><th>Value (ms)</th></tr></thead><tbody><tr><td>min</td><td>~180</td></tr><tr><td>max</td><td>~500</td></tr><tr><td>median</td><td>223.7</td></tr><tr><td>p95</td><td>~350</td></tr><tr><td>p99</td><td>~450</td></tr></tbody></table>	Metric	Value (ms)	min	~180	max	~500	median	223.7	p95	~350	p99	~450	 <table border="1"><caption>FCP (React + Node)</caption><thead><tr><th>Metric</th><th>Value (ms)</th></tr></thead><tbody><tr><td>min</td><td>~350</td></tr><tr><td>max</td><td>~1150</td></tr><tr><td>median</td><td>432.7</td></tr><tr><td>p95</td><td>~600</td></tr><tr><td>p99</td><td>~650</td></tr></tbody></table>	Metric	Value (ms)	min	~350	max	~1150	median	432.7	p95	~600	p99	~650
Metric	Value (ms)																								
min	~180																								
max	~500																								
median	223.7																								
p95	~350																								
p99	~450																								
Metric	Value (ms)																								
min	~350																								
max	~1150																								
median	432.7																								
p95	~600																								
p99	~650																								
<b>FID (First Input Delay) : délai entre la première action de l'utilisateur et la réaction du site</b>																									
 <table border="1"><caption>FID (Elixir)</caption><thead><tr><th>Metric</th><th>Value (ms)</th></tr></thead><tbody><tr><td>min</td><td>~0.6</td></tr><tr><td>max</td><td>~1.1</td></tr><tr><td>median</td><td>0.9</td></tr><tr><td>p95</td><td>~1.0</td></tr><tr><td>p99</td><td>~1.0</td></tr></tbody></table>	Metric	Value (ms)	min	~0.6	max	~1.1	median	0.9	p95	~1.0	p99	~1.0	 <table border="1"><caption>FID (React + Node)</caption><thead><tr><th>Metric</th><th>Value (ms)</th></tr></thead><tbody><tr><td>min</td><td>~10</td></tr><tr><td>max</td><td>~58</td></tr><tr><td>median</td><td>30.9</td></tr><tr><td>p95</td><td>~48</td></tr><tr><td>p99</td><td>~53</td></tr></tbody></table>	Metric	Value (ms)	min	~10	max	~58	median	30.9	p95	~48	p99	~53
Metric	Value (ms)																								
min	~0.6																								
max	~1.1																								
median	0.9																								
p95	~1.0																								
p99	~1.0																								
Metric	Value (ms)																								
min	~10																								
max	~58																								
median	30.9																								
p95	~48																								
p99	~53																								
<b>TTFB (Time To First Byte) : délai avant la réception du premier byte de la page</b>																									
 <table border="1"><caption>TTFB (Elixir)</caption><thead><tr><th>Metric</th><th>Value (ms)</th></tr></thead><tbody><tr><td>min</td><td>~120</td></tr><tr><td>max</td><td>~260</td></tr><tr><td>median</td><td>133</td></tr><tr><td>p95</td><td>~170</td></tr><tr><td>p99</td><td>~210</td></tr></tbody></table>	Metric	Value (ms)	min	~120	max	~260	median	133	p95	~170	p99	~210	 <table border="1"><caption>TTFB (React + Node)</caption><thead><tr><th>Metric</th><th>Value (ms)</th></tr></thead><tbody><tr><td>min</td><td>~220</td></tr><tr><td>max</td><td>~460</td></tr><tr><td>median</td><td>237.5</td></tr><tr><td>p95</td><td>~360</td></tr><tr><td>p99</td><td>~420</td></tr></tbody></table>	Metric	Value (ms)	min	~220	max	~460	median	237.5	p95	~360	p99	~420
Metric	Value (ms)																								
min	~120																								
max	~260																								
median	133																								
p95	~170																								
p99	~210																								
Metric	Value (ms)																								
min	~220																								
max	~460																								
median	237.5																								
p95	~360																								
p99	~420																								
<b>Longueur de la session (pour réaliser les 10 traits aléatoires)</b>																									
 <table border="1"><caption>Session Length (Elixir)</caption><thead><tr><th>Metric</th><th>Value (ms)</th></tr></thead><tbody><tr><td>min</td><td>~850</td></tr><tr><td>max</td><td>~1400</td></tr><tr><td>median</td><td>1,043.3</td></tr><tr><td>p95</td><td>~1150</td></tr><tr><td>p99</td><td>~1250</td></tr></tbody></table>	Metric	Value (ms)	min	~850	max	~1400	median	1,043.3	p95	~1150	p99	~1250	 <table border="1"><caption>Session Length (React + Node)</caption><thead><tr><th>Metric</th><th>Value (ms)</th></tr></thead><tbody><tr><td>min</td><td>~1300</td></tr><tr><td>max</td><td>~2000</td></tr><tr><td>median</td><td>1,380.5</td></tr><tr><td>p95</td><td>~1550</td></tr><tr><td>p99</td><td>~1650</td></tr></tbody></table>	Metric	Value (ms)	min	~1300	max	~2000	median	1,380.5	p95	~1550	p99	~1650
Metric	Value (ms)																								
min	~850																								
max	~1400																								
median	1,043.3																								
p95	~1150																								
p99	~1250																								
Metric	Value (ms)																								
min	~1300																								
max	~2000																								
median	1,380.5																								
p95	~1550																								
p99	~1650																								

Peu importe la métrique choisie, l'application Elixir est plus rapide que la version Node. Il faut tout de même noter que l'application Node contient en réalité deux applications et que le frontend doit donc contacter le backend. Cette communication moins directe pourrait donc être la cause des résultats moins bons de ce benchmark. Cependant, il s'agit d'un argument en faveur d'Elixir et plus particulièrement de Phoenix. En effet, il est de plus en plus courant de séparer les applications web en deux en utilisant des frameworks JavaScript de plus en plus complexes et loin d'être légers. Cet exemple nous démontre que l'approche plus « traditionnelle » qu'est cette version monolithe MVC a encore des avantages et une raison d'être.

## Bilan

Ce dernier chapitre sert de synthèse en reprenant les différentes attentes du langage et en vérifiant dans la pratique si elles ont été ou non vérifiées.

### Aspects positifs

L'argument de la performance est sans réelle contestation vérifié. En effet, il n'a même pas été possible de véritablement tester l'application dans ses derniers retranchements. De plus, il faut souligner que l'application Elixir est très légère en comparaison à la version Node. Lors du déploiement de la version Node, il a fallu la modifier pour que celle-ci puisse être déployée avec seulement les 256MB de mémoire disponible dans la version gratuite. Ce qui n'a pas été un souci du tout pour l'application Elixir. La faible latence annoncée a également pu être observée en comparant les différents graphiques de résultats du test de montée en charge.

Son écosystème web, plus particulièrement le framework Phoenix, est vraiment convaincant. Il est agréable à utiliser et vient avec de nombreuses solutions clés en main qui permettent au développeur d'être efficace très rapidement. Sa structure MVC permet également de ne pas être trop perdu en venant d'un autre langage. Elle est très similaire à ce que l'on pourrait retrouver sur des frameworks plus anciens et établis comme Ruby on Rails ou Laravel. Pour finir, sa gestion des Websockets est très complète et bien pensée et on remarque qu'il s'agit d'un point important dans leur communication. C'est en effet le premier élément mis en avant sur la page d'accueil du framework.

### Aspects négatifs

Le principal souci rencontré durant le développement du projet a été la taille de la communauté autour du langage. En effet, il a été compliqué (voire impossible) de trouver des exemples de benchmarks Websockets avec l'implémentation spéciale proposée par Phoenix. Ce qui aurait sûrement été plus simple avec une adoption plus large.

Pour finir, la prise en main du langage ne m'a pas été si aisée. Il avait été annoncé qu'Elixir avait l'objectif d'être aussi accessible que Ruby ou Python par exemple. De mon point de vue, ce n'est pas vraiment le cas. La syntaxe reste assez proche mais les concepts sont plutôt opposés. Utiliser un langage purement fonctionnel comme Elixir demande dans certains cas de repenser son modèle mental. Cependant, c'est notamment grâce à cette approche fonctionnelle que de telles performances sont possibles. Il est donc justifié de s'accrocher durant la prise en main plus ou moins compliquée.