HEIG VD
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Paradigme de concurrence

En utilisant Elixir

2025-06-12

Guillaume Dunant & Edwin Häffner
HEIG-VD
2025
ICSL

Table des matières

I. Implémentation et Architecture Logicielle

Introduction

Notre projet, **Space capitalism**, est un jeu dans lequel le but est de gagner un maximum de doublons galactiques ($dG), la monnaie du jeu. Pour cela, il est possible d'acheter des planètes ainsi que des robots qui vont pouvoir travailler sur ces planètes afin de récupérer des ressources comme du fer ou de l'or. Ensuite, il est possible de vendre ces ressources à la bourse pour gagner des $dG! Mais attention, les prix de la bourse varient au cours du temps, en bien ou en mal. Cela permet de pouvoir spéculer sur la valeur des ressources ou bien d'investir dans l'une des cryptomonnaies pour essayer d'en tirer un bénéfice. Des événements aléatoires interviennent également au cours d'une partie. Ceux-ci peuvent affecter la bourse, la réserve d'argent du joueur ou les robots d'une planète. Il y a aussi la taxe intergalactique ainsi que le coût de maintenance des robots qui sont prélevés automatiquement de manière périodique. Enfin, une série d'améliorations sont possibles afin de booster la récolte de certaines ressources.

Utilisation d'LLMs

Lors de la conception de ce projet et de ce rapport, des LLMs (Claude, Gemini) ont été utilisés. La partie frontend n'étant ni notre fort, ni le cœur de ce projet, nous avons décidé de les utiliser pour avoir une interface utilisateur agréable sans devoir perdre trop de temps, du temps crucial pour coder la partie métier de ce projet.

Ensuite, ils nous ont aidés pour la reformulation et l'amélioration de certaines parties du rapport.

Architecture générale de l'application Phoenix

Pour effectuer ce projet, nous avons choisi d'utiliser le framework Phoenix, qui permet de développer des applications web en utilisant le langage Elixir. Le choix de développer une application web s'est imposé car Phoenix représente le framework le plus populaire et mature de l'écosystème Elixir pour les interfaces graphiques. De plus, l'opportunité de créer une application full stack entièrement en Elixir était particulièrement attrayante pour explorer en profondeur le paradigme de concurrence dans notre contexte de jeu.

Structure MVC avec Phoenix Framework

Notre application suit l'architecture MVC (Model-View-Controller) de Phoenix, mais adaptée pour tirer parti du paradigme de concurrence d'Elixir. Nous avons structuré l'application en deux couches principales :

Backend concurrent (Model) : La partie backend implémente une architecture basée sur des processus Elixir communicant par messages. Elle gère toute la logique métier et les interactions inter-processus de notre jeu, exploitant pleinement le modèle de concurrence du paradigme Actor d'Elixir.

Frontend avec Phoenix LiveView (Controller/View unifié) : Le frontend utilise Phoenix LiveView, une technologie qui dépasse l'architecture MVC traditionnelle en unifiant Controller et View. Elle permet de créer des interfaces web interactives en temps réel, similaire à React mais côté serveur. Le fichier `game_live.ex` constitue le composant principal qui gère

à la fois l'état de l'interface utilisateur et les interactions. Il maintient en mémoire l'état du jeu (ressources, planètes, marché, statistiques VM) et se met à jour en demandant les nouvelles informations au backend, automatiquement toutes les 200ms.

Le template `game_live.html.heex` définit l'interface utilisateur qui comprend :

- Un tableau de bord des ressources de l'utilisateur en temps réel
- Une interface de gestion des planètes avec colonisation et gestion des robots
- Un marché boursier pour échanger des ressources
- Un laboratoire de recherche pour acheter des améliorations
- Un moniteur système temps réel affichant les métriques de la BEAM VM (processus actifs, mémoire, garbage collector, etc.)

Organisation des modules Elixir

Tous les modules pour le backend que nous avons créés pour ce projet se trouvent dans le dossier `lib>space_capitalism>components`. Tous ces modules sont des `Agent`, des `GenServer`, des `Supervisor` ou des `DynamicSupervisor`. Pour rappel, ce sont des comportements fournis par OTP. Les `Agent` et `GenServer` permettent de garder un état interne et d'envoyer / recevoir des messages. Les `Supervisor` et `DynamicSupervisor` permettent de démarrer ou de stopper d'autres processus et de les redémarrer en cas d'arrêt non prévu.

Dans notre backend, nous avons quatre `Supervisor` / `DynamicSupervisor` qui sont:

- GameSupervisor
- PlanetSupervisor
- ResourceSupervisor
- RobotDynSupervisor

Le reste des modules sont des `Agent` / `GenServer` :

- EventManager
- Planet
- Resource
- Robot
- StockMarket

En plus de cela, se trouve le module `UpdateManager` qui sert à regrouper les fonctions concernant les améliorations, mais qui n'a pas un comportement spécifique.

Implémentation du projet

Arbre de supervision

Afin d'organiser nos processus, nous avons utilisé le principe d'arbre de supervision mis en place par OTP[1]. À la racine de cet arbre se trouve le module `GameSupervisor` qui a pour unique but de démarrer les autres superviseurs et les serveurs qui ne dépendent pas d'un autre superviseur. La stratégie utilisée par tous les superviseurs est `one_for_one` qui permet de redémarrer uniquement le processus qui s'est arrêté.

```
def init(_) do
    # Children of the GameSupervisor
    children = [
```

[1]https://www.erlang.org/doc/system/design_principles.html

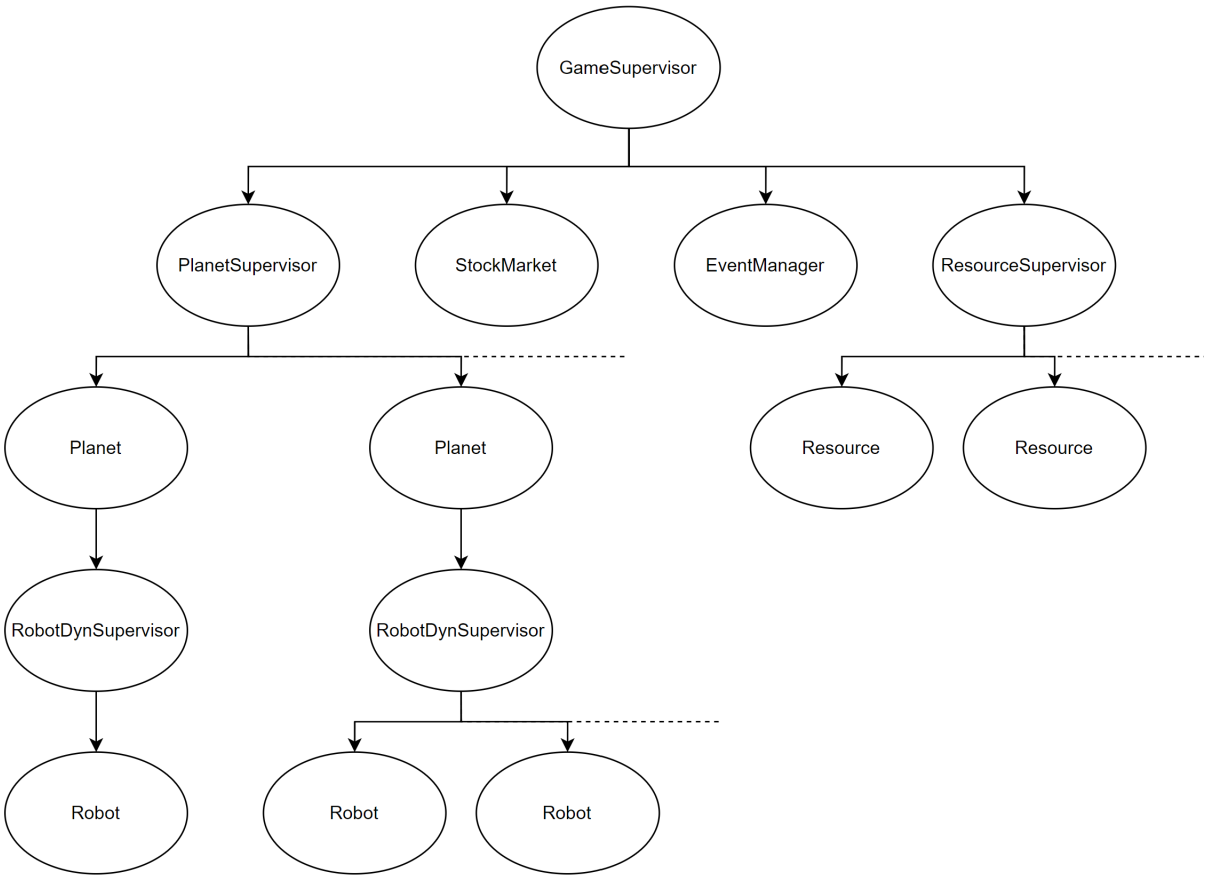
```
      ResourceSupervisor,
      PlanetSupervisor,
      StockMarket,
      EventManager
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
```

Ensuite, le `ResourceSupervisor` va démarrer un `Agent` `Resource` pour chacune des ressources du jeu et `PlanetSupervisor` va démarrer un `GenServer` `Planet` par planète disponible dans le jeu. Quand une planète est démarrée, elle lance également un `RobotDynSupervisor` qui a pour but de gérer les robots de la planète qui l'a démarré. Le `RobotDynSupervisor` ne lance pas d'autres processus en même temps que lui, mais pourra démarrer ou stopper les `GenServer` `Robot` quand nécessaire.

Grâce à ces superviseurs, on est assuré d'avoir nos processus toujours en vie, comme les superviseurs redémarrent leurs enfants automatiquement en cas d'arrêts. Organiser les superviseurs en arbre permet de plus de s'assurer que les superviseurs eux-mêmes soient ainsi surveillés par leur parent.

Figure 1 — Arbre de supervision de Space Capitalism



Gestion des ressources

Les ressources dans notre jeu sont simplement un nombre entier qui représente la quantité possédée par le joueur. Du fait de cet état simple, nous avons utilisé le comportement `Agent` d'OTP. Ce comportement met à disposition trois méthodes pour pouvoir accéder et/

ou modifier son état tout en garantissant sa cohérence en gérant les accès concurrents. Dans notre application, nous avons pu utiliser les trois fonctions pour répondre à nos besoins:

- `Resource.get` : cette fonction utilise `Agent.get` qui permet de retourner la valeur stockée dans l'état de l'agent.

```
def get(name) do
  Agent.get(name, fn count -> count end)
end
```

- `Resource.add` : cette fonction permet d'ajouter une certaine quantité à la valeur actuelle. Elle utilise la fonction `Agent.update` . Celle-ci prend en paramètre une autre fonction qui décrit comment mettre à jour la valeur. Cette fonction reçoit en paramètre la valeur de l'état actuel, ce qui permet d'éviter d'y accéder depuis ailleurs (avec un `get` par exemple) pour déterminer la valeur du nouvel état.

```
def add(name, amount) do
  Agent.update(name, fn count -> count + amount end)
end
```

- `Resource.remove` : cette fonction permet de retirer une certaine quantité à la valeur actuelle, comme par exemple lors d'un achat, on veut retirer l'argent. Cependant, avant de retirer la quantité, on aimerait pouvoir vérifier si la quantité possédée est suffisante. Par exemple, si le joueur veut dépenser 1000 $dG mais qu'il en possède seulement 500, la requête devrait échouer. Si cette vérification se fait en-dehors de l'agent (on obtient la valeur actuelle, on vérifie si la valeur est assez grande puis on retire la quantité désirée), il se peut qu'un autre processus ait demandé de diminuer la quantité et que donc notre vérification ne soit plus correcte. Pour répondre à ce besoin, les `Agent` mettent à disposition la fonction `Agent.get_and_update` . Cette fonction permet de combiner les deux fonctions précédentes en modifiant la valeur contenue dans l'état et de renvoyer une valeur à l'appelant. Dans le cas de `Resource.remove` , la fonction vérifie si la quantité possédée est suffisante. Si c'est le cas, la mise à jour est faite et la fonction retourne `:ok` . Si ce n'est pas le cas, la mise à jour n'est pas faite et l'appelant est informé que l'opération a échoué en retournant l'atome `:error` . La vérification et la modification sont faites de manière atomique, sans risque d'accès concurrent.

```
def remove(name, amount) do
  Agent.get_and_update(name, fn count ->
    if count >= amount do
      # Successful removal
      {{:ok, count - amount}, count - amount}
    else
      # Cannot remove more than available
      {{:error, :insufficient_resources}, count}
    end
  end)
end
```

Chaque ressource est démarrée depuis le `ResourceSupervisor` , avec en paramètre la quantité que le joueur a en début de partie ainsi qu'un atome représentant le nom de la ressource.

Gestion des robots

Chaque robot créé au cours de la partie est un processus. Ceux-ci ont pour unique but de lancer une boucle infinie où ils vont ajouter une certaine quantité de ressource à la réserve du

joueur, puis atteindre le prochain moment où ils vont répéter cette action. Leur comportement utilise les `GenServer` d'OTP. Ceux-ci sont similaires aux `Agent` du fait qu'ils gèrent un état interne, mais leur implémentation se fait à l'aide de callbacks aux différents messages qu'ils peuvent recevoir et non avec les méthodes `get`, `update` ou `get_and_update`. Cela permet d'utiliser la fonction `Process.send_after` qui permet d'envoyer un message à un processus après une certaine durée d'attente.

```
Process.send_after(self(), :work, state[:time])
```

Lors de l'initialisation du `GenServer` du robot, on appelle cette fonction qui va permettre d'envoyer un message contenant `:work` au processus lui-même. Ensuite, il faut implémenter le callback qui va réagir à la réception de ce message. Cela se fait à l'aide de la méthode `handle_info`.

```
# Handle the work loop
  @impl true
  def handle_info(:work, state) do
    # Do the work ...

    Process.send_after(self(), :work, next_work_time)
    {:noreply, state}
  end
```

Le callback va effectuer l'action du robot, puis se rappeler de manière récursive en réutilisant la fonction `Process.send_after`.

Les robots sont créés et supprimés depuis le `RobotDynSupervisor`. Ce superviseur permet de bien vérifier que les robots ne s'arrêtent pas et de les redémarrer en cas de besoin. Pour les stopper, le superviseur dynamique va simplement appeler la fonction `terminate_child` qui permet d'arrêter volontairement un processus enfant. Quand le processus essaiera d'envoyer le message `:work` pour continuer la boucle de travail, comme le destinataire n'existe plus, le message sera ignoré et la boucle ainsi stoppée.

En plus des superviseurs, il y a également une ressource robot qui permet de gérer facilement le nombre global de robots pour l'affichage dans l'interface du jeu.

Gestion des autres GenServer

En-dehors des ressources et des robots, les autres modules ont une implémentation et une utilité assez similaire. Ils gèrent un état (le coût ou la ressource produite pour les planètes, les valeurs d'achat/revente des ressources pour la bourse, etc...) et mettent à disposition des fonctions publiques pour interagir avec cet état. Ces fonctions publiques vont à chaque fois avoir pour unique but d'envoyer un message au `GenServer` avec comme contenu l'action à effectuer et les éventuels paramètres. Toute la logique métier se trouve ensuite dans les callbacks des messages. Faire ainsi permet aux autres modules d'interagir facilement avec ce module avec les fonctions publiques tout en garantissant la cohérence interne en encapsulant la logique dans les callbacks pour garantir l'exécution séquentielle grâce aux messages.

LiveView

L'utilisation de LiveView était très intéressante dans le projet. Phoenix LiveView permet de créer un pont direct entre la concurrence serveur et l'interface utilisateur. Elle permet d'avoir une mise à jour des informations sur la page sans devoir la recharger entièrement. LiveView

envoie à l'utilisateur uniquement des diffs HTML pour effectuer cette mise à jour de façon efficiente.

Communication asynchrone temps réel :

Ici, on a mis en place un système de mise à jour des informations liées à l'interface directement dans notre LiveView. Toutes les 200ms, un timer envoie un message `:update_display` au processus LiveView (référencé par `self()`), qui déclenche l'exécution de la fonction `handle_info/2` correspondante.

Flow d'execution d'un processus Live View

```
:timer.send_interval(200, self(), :update_display)
# ↓ Toutes les 200ms
# Message {:update_display} → Mailbox du processus LiveView
# ↓ Traitement asynchrone
def handle_info(:update_display, socket) do
  # Mise à jour des données
end
```

Ce système permet d'avoir une mise à jour des informations utilisateur complètement asynchrone, créant un pont entre la concurrence du backend et l'interface utilisateur.

Communication événementielle via PubSub :

Pour afficher des informations du backend vers le frontend de façon complètement asynchrone, sans polling ni requêtes de mise à jour, nous avons utilisé le système `PubSub.broadcast` qui diffuse des messages. Le processus LiveView s'abonne à ces messages via `PubSub.subscribe` et les affiche automatiquement dès leur réception. Nous avons utilisé cette fonctionnalité pour la gestion des événements galactiques aléatoires.

Aspects techniques spécifiques

Démarrer l'application

Il est possible de démarrer l'application soit localement en ayant Elixir et Phoenix d'installés ou bien d'utiliser Docker. Le guide détaillé se trouve dans le fichier `README.md`.

Déploiement et scalabilité

Pour le moment, l'application n'est pas déployée. Cette décision est entièrement volontaire. En effet, il n'y a pas de mécanisme de sessions mis en place et toute personne se connectant sur le serveur aurait accès à la même partie en cours.

Cette approche mono-utilisateur était adaptée à nos objectifs car nous avons utilisé Phoenix Framework pour sa popularité et sa gestion graphique, non pour ses capacités web multi-utilisateurs avec les websockets. L'objectif était d'explorer le paradigme de concurrence d'Elixir dans un contexte applicatif complet plutôt que de créer une véritable application web distribuée. Le développement d'un système de sessions multi-utilisateurs aurait détourné encore plus notre attention du paradigme de concurrence, qui était le cœur de ce projet…

II. Retour d'Expérience

Expérience avec le paradigme de concurrence

Ce qui est impressionnant avec Elixir, c'est qu'on oublie presque qu'on utilise le paradigme de concurrence. Dans notre application, tout passe par messages, et l'endroit clé de la concurrence est le compteur global des ressources de l'utilisateur. En utilisant l'architecture idiomatique d'Elixir avec les patterns Acteur/Agent, il n'y a presque aucun moyen d'avoir des problèmes de concurrence.

Tous les problèmes de sections critiques partagées et de ressources qui peuvent être modifiées par plusieurs processus à la fois ne constituent plus un problème, car tout se fait de façon séquentielle dans la boîte aux lettres du processus, éliminant ainsi les conditions de course.

Défis rencontrés et solutions adoptées

Ce qui reste complexe est l'adoption du modèle de conception. On ne peut pas écrire cette application comme on l'aurait écrite en Java ou en C++. Devoir communiquer entre processus en utilisant des messages est un peu complexe au début, mais on s'y habitue rapidement.

Il y a aussi un défi avec l'intégration IDE. Dès qu'on commence à créer nos propres modules et à devoir les utiliser, les IDEs n'ont pas l'air d'apprécier parfaitement Elixir. Il est quasi impossible d'avoir une vue claire de la structure de nos modules…

Comparaison avec d'autres paradigmes de concurrence

De tous les langages utilisant le paradigme de concurrence, Elixir est sûrement le plus simple à utiliser. Si on compare avec ce qu'on a pu utiliser, c'est-à-dire des librairies pour la concurrence en C++ (mutex, sémaphores, etc.) ou les channels en Go, la différence est grande !

Avec C++, il faut constamment gérer les verrous, s'assurer qu'on n'a pas de deadlocks, et la moindre erreur peut planter l'application entière et déboguer ces problèmes est horrible. Avec Go, bien que les channels soient élégants, on doit encore réfléchir à la synchronisation et aux patterns de communication, devoir mettre en place une goroutine qui gère les messages entrants de façon synchrone à la main…

Avec Elixir, une fois qu'on passe outre la manière un peu originale d'organiser le code, la gestion de la concurrence devient presque transparente. Pas de verrous à gérer, pas de memory leaks possibles, et si un processus plante, les autres continuent tranquillement. C'est vraiment supérieur chez Elixir/Erlang.

Retour sur le langage Elixir

Aspects fonctionnels et syntaxe

Le langage ressemble quelque peu à Haskell/Scala par sa programmation fonctionnelle. La logique reste similaire mais la syntaxe diffère. Ce qui change le plus par rapport aux langages typés statiquement que nous connaissons (Java, C++, Go) est le fait qu'on utilise du typage dynamique, comme JavaScript et Python. Ce qui peut rendre l'écriture du code un peu difficile avec des erreurs de typage qui ne se révèlent seulement à l'exécution du code…

L'immutabilité est omniprésente en Elixir. Toutes les structures de données sont immutables par défaut, ce qui élimine les problèmes de concurrence liés aux mutations partagées. Dans notre projet, cela se manifeste par des mises à jour d'état comme `%{state | robot_count: new_count}` qui créent de nouvelles structures plutôt que de modifier l'existante.

Pattern Matching

Elixir étant un langage fonctionnel, nous avons beaucoup utilisé le pattern matching. Plutôt que d'utiliser des if/else ou des switch, nous pouvons déstructurer directement les données, rendant le code vraiment plus simple à lire :

```
case Resource.remove(:dG, total_cost) do
  {:ok, remaining} -> # Transaction réussie
  {:error, :insufficient_funds} -> # Pas assez d'argent
end
```

Atoms et communication par messages

Les atoms (`:iron` , `:dG` , `:nextEvent`) sont des sortes d'étiquettes uniques en Elixir qui remplacent les énumérations ou constantes d'autres langages. Ils servent d'identifiants rapides pour les messages entre processus et les types de ressources dans notre jeu. Comme Elixir ne stocke chaque atom qu'une seule fois en mémoire et les compare instantanément, ils sont parfaits pour la communication entre les différents processus du jeu.

On les utilise massivement pour les messages pour ensuite les diriger vers la fonction adéquate.

Pipe Operator et lisibilité

L'opérateur pipe `|>` transforme l'écriture du code en rendant les transformations de données naturelles à lire :

```
socket
|> assign(:resources, new_resources)
|> put_flash(:info, message)
|> then(&{:noreply, &1})
```

Cela évite l'imbrication excessive de fonctions qu'on retrouve dans d'autres langages fonctionnels.

Défis d'adaptation

L'adoption d'Elixir demande un changement de paradigme. On ne peut pas aborder les problèmes comme en Java ou C++. Il faut penser en termes de transformation de données et de flux de messages plutôt qu'en objets mutables.

La syntaxe reste accessible, même si apprendre le langage ne se fait pas en 10 minutes, c'était tout de même abordable !

Expérience avec Phoenix Framework

L'utilisation du framework Phoenix était très agréable. Une fois qu'on arrive à dépasser la structure quelque peu originale du framework (notamment le routing et la manière de l'utiliser qui est un peu déroutante au premier abord), l'utilisation de Phoenix se fait sans accrocs.

Tout y est très facilité : la communication temps réel via LiveView pour les mises à jour de l'interface utilisateur, l'utilisation des différents modules, etc. On n'a pas vraiment

l'impression de travailler séparément sur un frontend et un backend, mais directement dans une application full stack. De plus, le hot reloading est un vrai atout, on peut tout changer dans notre code et l'application reste très réactive, elle se rebuild aussi très rapidement.

Rien à redire.

III. Conclusion

Ce projet nous a permis d'explorer en profondeur le paradigme de concurrence à travers Elixir et Phoenix, en développant un jeu simulant des communications entre de nombreux processus concurrents. Cette expérience a été particulièrement enrichissante car nous avons pu découvrir un langage conçu nativement pour la concurrence!

Apports du paradigme de concurrence

Elixir, avec son modèle Actor et son architecture OTP, démontre à quel point la concurrence peut être naturelle quand elle est intégrée au cœur du langage. Le fait que tout passe par messages élimine les problèmes classiques de synchronisation (sections critiques, conditions de course) qu'on retrouve dans d'autres langages comme C++. On oublie presque qu'on utilise la concurrence tellement c'est transparent.

L'architecture avec les superviseurs offre également une grande tolérance aux pannes, permettant à chaque processus de redémarrer indépendamment sans affecter l'ensemble du système, même si dans notre projet, cette fonctionnalité n'a pas vraiment été utilisée.

Retour d'expérience

L'expérience de développement avec Phoenix a été très positive. L'intégration entre le backend concurrent et l'interface utilisateur via LiveView crée une application fluide et réactive. Le développement full stack en Elixir offre une cohérence appréciable, même si l'adoption du modèle de conception orienté messages demande un certain temps d'adaptation.

Les quelques défis rencontrés (courbe d'apprentissage, outils de développement) sont largement compensés par la simplicité et l'élégance du modèle de concurrence d'Elixir comparé à d'autres approches plus traditionnelles.

Au final c'était un projet très intéressant et c'est avec plaisir que nous réutiliserons Elixir pour d'autres projets.