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

Paradigme de concurrence

En utilisant Elixir

2025-06-11

Guillaume Dunant & Edwin Häffner
HEIG-VD
2025
ICSL

Table des matières

I. Implémentation et Architecture Logicielle

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) : Le frontend utilise Phoenix LiveView, une technologie qui permet de créer des interfaces web interactives en temps réel, d'une même façon que React. Le fichier `game_live.ex` constitue le contrôleur principal qui gère 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.

Ensuite nous avons la partie **View** avec le template `game_live.html.heex` qui définit l'interface utilisateur qui comprends :

- 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 créé pour ce projet ce retrouve dans la dossier `lib>space_capitalism>components` . Tous ces modules sont soient 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 de transmettre des messages. Les `Supervisor` et `DynamicSupervisor` permettent de démarrer ou de stopper des 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 concernants les améliorations mais qui n'a pas un comportement spécific.

Implémentation du paradigme de concurrence

Arbre de supervision

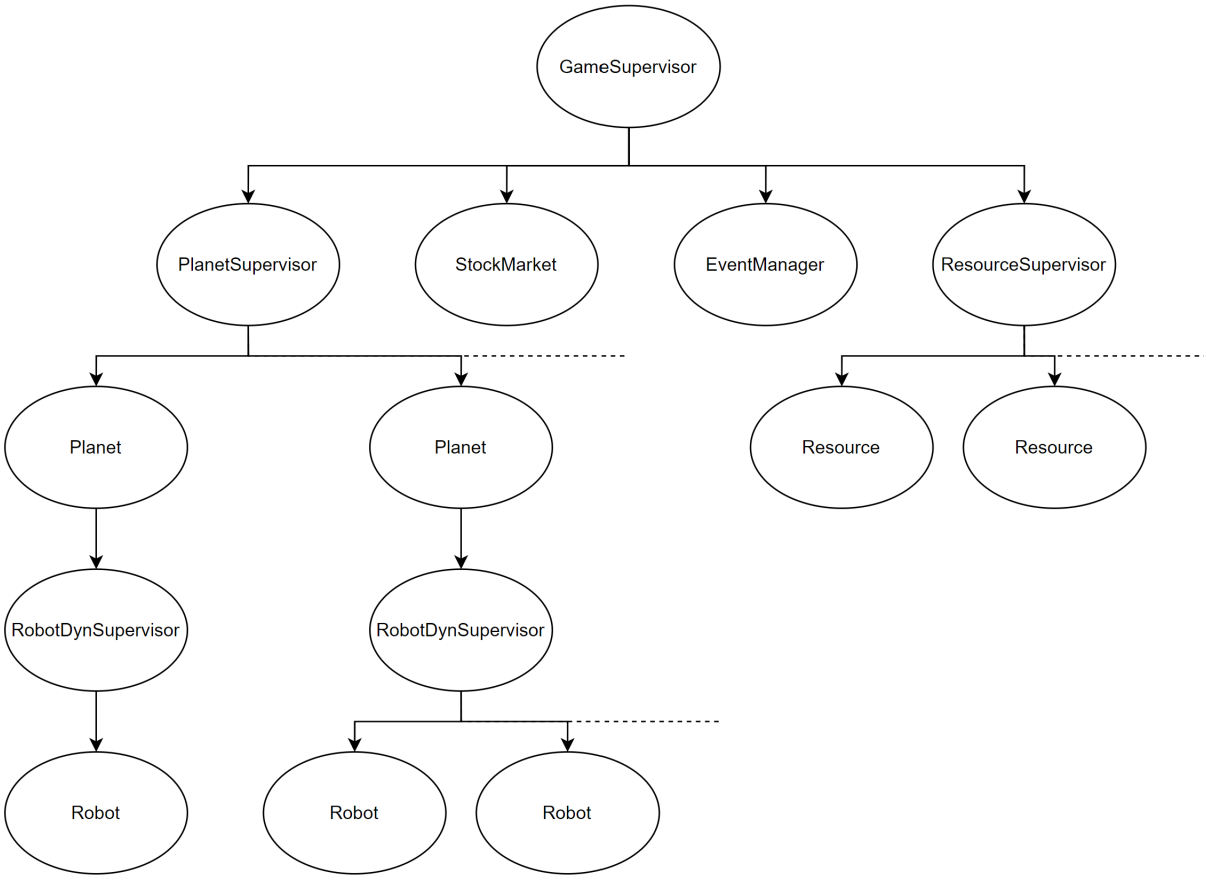
Afin d'organiser nos processus, nous avons utiliser le principe d'arbre de supervision mis en place par OTP. À la racine de cet arbre ce trouve le module `GameSupervisor` qui a pour unique but de démarer les autres superviseurs et les serveurs qui ne dépendent pas d'un autre superviseur. La stratégie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

```
def init(_) do
    children = [
      ResourceSupervisor,
      PlanetSupervisor,
      StockMarket,
      EventManager
    ]

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

Ensuite, le `ResourceSupervisor` va démarre un `Agent` `Resource` pour chacune des ressources du jeu et `PlanetSupervisor` va démarre 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 démarrer. Le `RobotDynSupervisor` ne démarre pas d'autre processus en même temps que lui, mais pourra démarrer ou stopper les `GenServer` `Robot` quand nécessaire.

Figure 1 — Arbre de supervision



Utilisation des GenServers

Supervision et tolérance aux pannes

Communication inter-processus par messages

LiveView

De plus, 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
```

```elixir
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

Configuration de l'application Phoenix

Pour lancer l'application Phoenix, il faut en premier lieu installer Elixir (version 1.14 ou supérieure) et Phoenix Framework sur votre machine, tuto ici : https://hexdocs.pm/phoenix/installation.html. Ensuite, les étapes pour démarrer l'application sont simples :

1. Se placer dans le dossier `space_capitalism`
2. Installer les dépendances avec `mix deps.get`
3. Compiler les dépendances avec `mix deps.compile`
4. Lancer le serveur avec `mix phx.server`

L'application devient alors accessible sur `http://localhost:4000`. Phoenix se charge automatiquement de compiler les assets et de démarrer tous les processus nécessaires grâce à la configuration OTP de notre application.

Déploiement et scalabilité

Pour le moment, l'application n'est pas déployée. La raison principale de ce manque de déploiement réside dans la façon dont nous avons développé l'application. En effet, il n'y a pas de mécanisme de sessions mis en place et toute personne se connectant sur le serveur aura accès à la même partie en cours.

Ce n'est pas vraiment un problème pour nous 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.

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.

Avantages observés dans la pratique

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 debugger 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 go routine qui gère les messages entrant 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 font seulement à l'execution 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 language fonctionnel, nous avons beaucoup utilisée 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 dirigers 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 language 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 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 des milliers de processus. 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 utilisé.

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.