

# Paradigme de concurrence

## En utilisant Elixir

2025-06-09

**Guillaume Dunant & Edwin Häffner**

HEIG-VD

2025

*ICSL*

# Table des matières

<b>I. Implémentation et Architecture Logicielle .....</b>	<b>3</b>
Architecture générale de l'application Phoenix .....	3
Structure MVC avec Phoenix Framework .....	3
Organisation des modules Elixir .....	3
Implémentation du paradigme de concurrence .....	4
Arbre de supervision .....	4
Utilisation des GenServers .....	5
Supervision et tolérance aux pannes .....	5
Communication inter-processus par messages .....	5
LiveView .....	5
Aspects techniques spécifiques .....	6
Configuration de l'application Phoenix .....	6
Déploiement et scalabilité .....	6
<b>II. Retour d'Expérience .....</b>	<b>7</b>
Expérience avec le paradigme de concurrence .....	7
Avantages observés dans la pratique .....	7
Défis rencontrés et solutions adoptées .....	7
Comparaison avec d'autres paradigmes de concurrence .....	7
Retour sur le langage Elixir .....	7
Courbe d'apprentissage et adaptation .....	7
Points forts du langage découverts .....	7
Limitations rencontrées .....	7
Écosystème et communauté .....	7
Expérience avec Phoenix Framework .....	7
Facilité de développement web .....	7
Performance et réactivité .....	8
Outils de développement et debugging .....	8
Intégration avec l'écosystème Elixir/OTP .....	8
<b>III. Conclusion .....</b>	<b>9</b>
Synthèse des apprentissages .....	9
Pertinence du paradigme de concurrence pour ce type de projet .....	9
Perspectives d'amélioration et évolutions futures .....	9
Recommandations pour de futurs projets similaires .....	9

## 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 communiquant 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 se retrouvent dans le dossier `lib>space_capitalism>components`. Tous ces modules sont soit 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 concernant les améliorations mais qui n'a pas un comportement spécifique.

## Implémentation du paradigme de concurrence

### Arbre de supervision

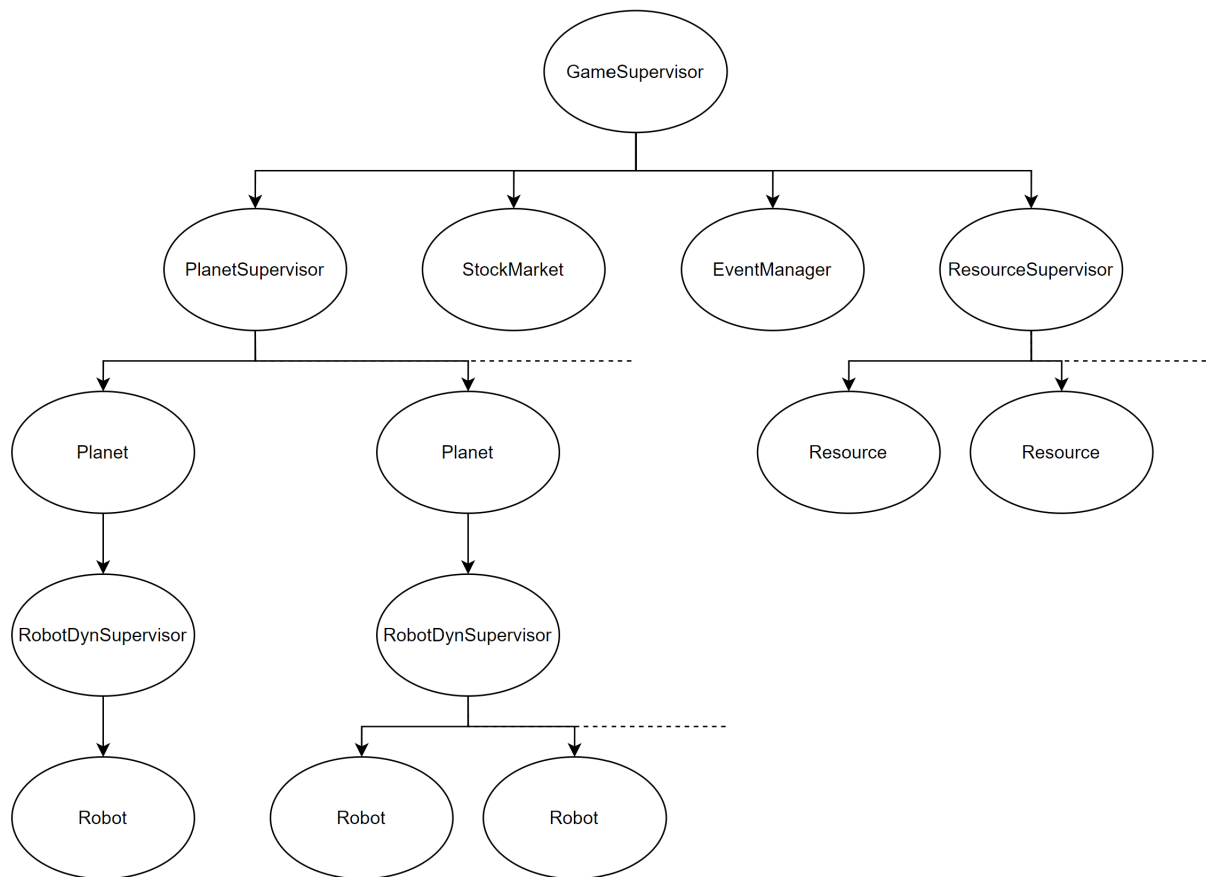
Afin d'organiser nos processus, nous avons utilisé le principe d'arbre de supervision mis en place par OTP. À la racine de cet arbre se trouve le module `GameSupervisor` qui a pour unique but de démarrer les autres superviseurs et les serveurs qui ne dépendent pas d'un autre superviseur. La stratégie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

```
def init(_) do
  children = [
    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 démarrée. 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 efficace.

#### 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 (référéncé par `self()`), qui déclenche l'exécution de la fonction `handle_info/2` correspondante.

#### Flow d'exécution 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**

### **Configuration de l'application Phoenix**

#### **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.

### Retour sur le langage Elixir

#### Courbe d'apprentissage et adaptation

Le langage ressemble quelque peu à Haskell par sa programmation fonctionnelle. La logique reste similaire mais la syntaxe diffère.

#### Points forts du langage découverts

#### Limitations rencontrées

#### Écosystème et communauté

### Expérience avec Phoenix Framework

#### Facilité de développement web

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 accroc.

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.

**Performance et réactivité**

**Outils de développement et debugging**

**Intégration avec l'écosystème Elixir/OTP**



### **III. Conclusion**

**Synthèse des apprentissages**

**Pertinence du paradigme de concurrence pour ce type de projet**

**Perspectives d'amélioration et évolutions futures**

**Recommandations pour de futurs projets similaires**