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

Paradigme de concurrence

En utilisant Elixir

2025-06-08

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

Gestion des processus et de la concurrence

Implémentation du paradigme de concurrence

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 `:updateDisplay` 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(), :updateDisplay)
# ↓ Toutes les 200ms
# Message {:updateDisplay} → Mailbox du processus LiveView
# ↓ Traitement asynchrone
def handle_info(:updateDisplay, 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

Avantages observés dans la pratique

Défis rencontrés et solutions adoptées

Comparaison avec d'autres paradigmes de concurrence

Retour sur le langage Elixir

Courbe d'apprentissage et adaptation

Le langage

Points forts du langage découverts

Limitations rencontrées

Écosystème et communauté

Expérience avec Phoenix Framework

Facilité de développement web

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