

Betunfair

Álvaro Plaza CArro

Agenda

Introducción

Requerimientos del sistema

Persistencia

Arquitectura del sistema

Escalabilidad

Tests

Conclusión





Introducción

El propósito de este proyecto final es la creación de un sistema de intercambio de apuestas escalable, desarrollado mediante el uso de Elixir, un lenguaje que se destaca por su tolerancia a fallos y sus capacidades de concurrencia. La funcionalidad principal de la plataforma estará encapsulada en un módulo de Elixir, que integrará las funciones esenciales necesarias para gestionar las apuestas, administrar las cuentas de los usuarios y emparejar apuestas contrapuestas.

Requerimientos del sistema

Base de datos

- **PostgreSQL:** Se utilizará como el sistema de gestión de bases de datos. PostgreSQL es robusto, escalable y maneja eficazmente transacciones concurrentes, lo que es fundamental para el sistema de apuestas

Elixir

- **Mix:** Esta es la herramienta de construcción y gestión de proyectos de Elixir. Usaremos Mix para gestionar nuestras dependencias, correr pruebas y más.
- **Ecto:** Ecto es una base de datos y lenguaje de consulta para Elixir. Nos ayudará a interactuar con nuestra base de datos PostgreSQL, proporcionando un API limpio y eficiente.
- **GenServer:** Un comportamiento del servidor genérico proporcionado por Elixir que nos ayudará a implementar servidores en nuestra aplicación. Esta será una parte fundamental de la arquitectura, ya que GenServer nos permitirá manejar el estado y la concurrencia de una manera segura y eficiente.

Persistencia de Datos con Ecto

- **Abstracción de la Base de Datos:** Ecto proporciona una interfaz común para diferentes sistemas de bases de datos, permitiendo el uso de distintas bases de datos con cambios mínimos en el código.
- **Integración con Changesets:** Changesets en Ecto permiten realizar un seguimiento de los cambios en nuestros datos antes de aplicarlos a la base de datos. Esto nos proporciona una validación de datos eficiente y control sobre cuándo y cómo se aplican los cambios.
- **Consultas Componibles:** Ecto utiliza su propio lenguaje de consultas, que permite componer consultas de una manera legible y segura.

Arquitectura del Sistema

- **Usuarios (Users):** Cada usuario tiene un ID único y un nombre. Los usuarios mantienen un balance, que puede aumentar o disminuir a medida que se realizan apuestas. Los usuarios pueden tener múltiples apuestas asociadas a ellos.
- **Apuestas (Bets):** Cada apuesta pertenece a un usuario y a un mercado. Incluye información sobre la cantidad de la apuesta, las probabilidades, el tipo de apuesta (compra o venta), y el estado de la apuesta.
- **Mercados (Markets):** Cada mercado tiene un nombre único y una descripción. Los mercados mantienen un estado que puede cambiar con el tiempo, y también tienen un resultado que se determina una vez que el mercado se resuelve. Las apuestas se realizan en relación a un mercado.
- **Emparejamientos (Matches):** Cada emparejamiento representa el hecho de que una apuesta lay y una apuesta back han sido emparejadas. Incluye la cantidad que ha sido emparejada entre estas dos apuestas

Escalabilidad y Confiabilidad

- **OTP:** El sistema utiliza GenServer, uno de los comportamientos que proporciona OTP. GenServer es un módulo que abstrae las funcionalidades para crear un servidor genérico concurrente, proporcionando una infraestructura sólida y robusta para nuestra aplicación. Permite crear funciones que manejan llamadas sincrónicas (`handle_call`), asincrónicas (`handle_cast`) y mensajes no esperados (`handle_info`), lo que facilita la construcción de una aplicación concurrente. En el código todas las operaciones son ejecutadas mediante llamadas al GenServer.

Escalabilidad y Confiabilidad

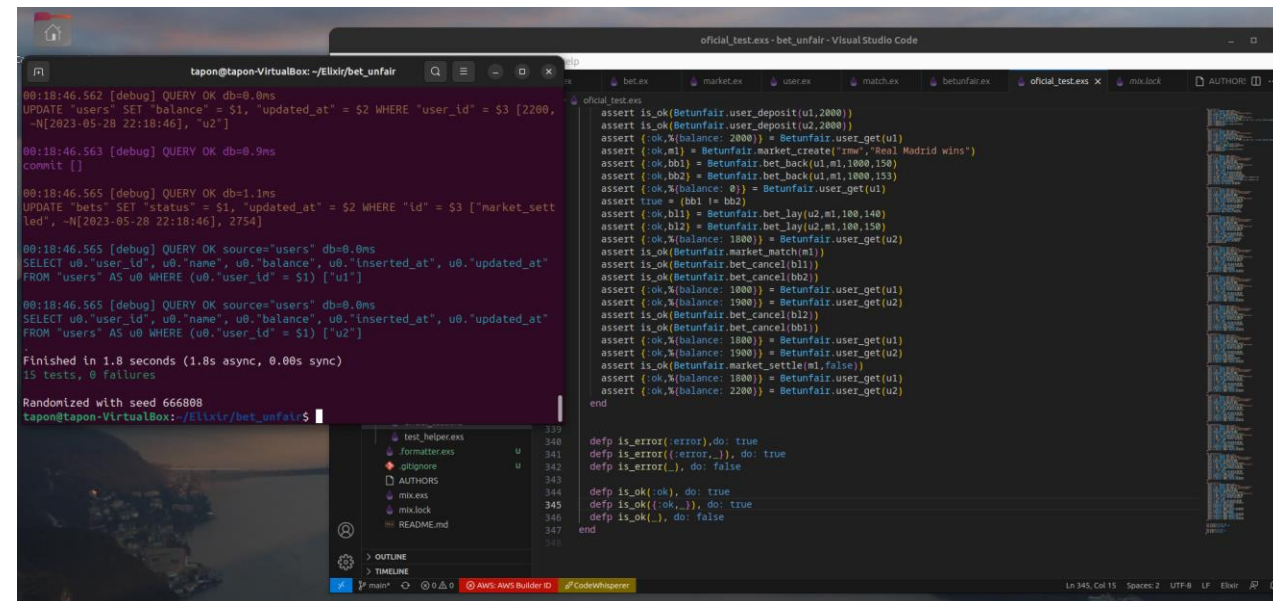
- **Ecto:** Ecto es un lenguaje de consultas integrado y un proyecto de adaptador de base de datos en Elixir. Nos permite manejar transacciones de manera muy eficiente y segura. En nuestro código, las transacciones se manejan mediante la función `Repo.transaction`, que nos asegura que todas las operaciones en la base de datos se realizan de forma atómica. Esto significa, si alguna parte de la transacción falla, todas las operaciones se revertirán, manteniendo así la integridad de nuestros datos.

Testing

Durante el desarrollo de nuestro programa, hemos implementado una serie de pruebas unitarias usando ExUnit.

En un principio las pruebas se centraron en controlar las entradas y salidas en la base de datos. Comprobar los distintos algoritmos desarrollados. Hasta poco a poco cumplir todos los tests de betunfair_test_exs

Captura tests



The screenshot shows a development environment with a terminal window on the left and a VS Code editor on the right. The terminal displays the output of running tests for the 'betunfair' application. It shows several SQL queries being executed, their results, and a final summary indicating that all tests passed. The VS Code editor shows the 'official_test.exs' file, which contains ExUnit test cases for various functions like 'user_deposit', 'market_create', 'bet_cancel', and 'market_settle'.

```
00:18:46.562 [debug] QUERY OK db=0.0ms
UPDATE "users" SET "balance" = $1, "updated_at" = $2 WHERE "user_id" = $3 [2200,
~N[2023-05-28 22:18:46], "u2"]

00:18:46.563 [debug] QUERY OK db=0.9ms
commit []

00:18:46.565 [debug] QUERY OK db=1.1ms
UPDATE "bets" SET "status" = $1, "updated_at" = $2 WHERE "id" = $3 ["market_sett
led", ~N[2023-05-28 22:18:46], 2754]

00:18:46.565 [debug] QUERY OK source="users" db=0.0ms
SELECT u0."user_id", u0."name", u0."balance", u0."inserted_at", u0."updated_at"
FROM "users" AS u0 WHERE (u0."user_id" = $1) ["u1"]

00:18:46.565 [debug] QUERY OK source="users" db=0.0ms
SELECT u0."user_id", u0."name", u0."balance", u0."inserted_at", u0."updated_at"
FROM "users" AS u0 WHERE (u0."user_id" = $1) ["u2"]

Finished in 1.8 seconds (1.8s async, 0.00s sync)
15 tests, 0 failures

Randomized with seed 666908
tapon@tapon-VirtualBox: ~/elixir/bet_unfair$
```

```
defp is_error(error), do: true
defp is_error(:error, _), do: true
defp is_error(_, _), do: false

defp is_ok(ok), do: true
defp is_ok(:ok, _), do: true
defp is_ok(_, _), do: false

end
```

Conclusión

Un extenso proceso de desarrollo orientado a crear un proyecto que cumpliera con todos los requisitos y pruebas. El reto principal fue la comprensión e implementación de los algoritmos para emparejar y resolver apuestas, que requirió un considerable esfuerzo.

Un área de mejora es la optimización del rendimiento al manejar un gran número de elementos. Por ejemplo, la función `market_bets` puede devolver todas las apuestas en un mercado, que podría ser un número considerable. En la versión actual del proyecto, hemos utilizado `Enum.map(& &1.id)` pero reconocemos que puede haber soluciones más eficientes disponibles, como las transmisiones perezosas (lazy streams) descritas en la documentación de Elixir. Implementar tal característica permitiría un mejor rendimiento al no necesitar inspeccionar todas las apuestas devueltas.