1 The Betting Exchange Platform Proposal

A betting exchange platform is a marketplace where users can place bets against each other, rather than against a bookmaker. Betfair Exchange and Smarkets[1] are popular examples of such a platform. In a betting exchange, users can either *back* a certain outcome (betting that an event will happen) or *lay* the same outcome (betting that the event will not happen). This allows users to trade positions and manage risk, leading to a more engaging and dynamic betting experience.

1.1 Users and Wallets

Users sign up for an account on the exchange and can deposit and withdraw funds.[2]

1.2 Events

Users can browse the available *events* (also known as markets) that people can place bets on. These events might be sports events as well as other activities such as political elections. Events or markets are typically registered and managed by the platform's operators or administrators. After the event has concluded, the market is *settled* by determining the winning bets based on the actual outcomes of the event. The platform then updates the users' account balances accordingly and closes the market.

1.3 Betting

Users can either back or lay an event, resulting in a backing bet, or a lay bet. For a backing bet to be "accepted, it must be *matched* with an acceptable lay bet. Vice versa, a lay bet must be matched with a backing bet. Here's an explanation of how it works:

Backing an Event:

When you *back* an event, you are betting that the event will happen. For example, you might back the event "Real Madrid to win the *clásico* against Barcelona". A backing bet specifies a *stake* and an *odds*. The *stake* is the amount of money you are willing to risk (the backing stake). The *odds* offered for backing an event represent the potential winnings if your bet is successful, which is calculated as *stake* $\times$ *odds*.[3]

For example, suppose that you are willing to risk $20€$ to back "Real Madrid to win the *clásico* against Barcelona", and you specify an odds of 1.5. Suppose moreover that some other person *lays* a *matching* bet against Real Madrid winning, then first your $20€$ (the stake) is removed from your account. After the *clásico* finishes the market is *settled*. If Real Madrid does win your account will receive $20 \times 1.5€ = 30€$ (a gain of $10€$). But if Real Madrid draws or looses, you will have lost your original stake of $20€$.

Laying an Event:

When you *lay* an event, you are betting that the event will *not* happen, i.e., in our example, laying "Real Madrid to win the *clásico* against Barcelona", is betting that F. C. Barcelona draws or wins the *clásico*. In other words, you are acting as the bookmaker and taking on the risk of paying out the winnings if the event occurs.

A lay bet specifies a *stake* (lay stake) and an *odds* too. The *stake* is the amount of money you are willing to risk, exactly as in a backing bet.

[1] `https://www.betfair.es` and `https://smarkets.com`, respectively.
[2] Obiously you will simulate it in your implementation!
[3] Typically minus some small percentage of money which goes to the operators of the exchange platform itself.

If the event does not occur, you win the bet and keep the stake of the backer. If the event does happen, you must pay the backer their winnings, calculated as their stake multiplied by the odds, minus the stake itself.

Continuing with the example above, i.e., a backing bet with stake $20€$ and odds 1.5, can be matched with a lay bet with a stake at least $10€$ and an odds that is 1.5 or greater.

If Real Madrid wins the *clásico* the lay better needs to pay out $20 * 1.5 - 20€ = 10€$(the gain of the backing better minus his stake). If Real Madrid does not win then the lay better gains the stake of the backing better, i.e., $20€$.

As can be seen, it is advantagous for the lay better to specify as low odds as possible. On the other hand, maybe no backing better will accept such a low odds.

Matches can be partial too. For example, if backing and lay better agrees on an odds of 3.0, with both stakes $20€$, then the lay better has to pay out $20 * 3.0 - 20€ = 40$ euros. But the lay stake is only $20€$ – impossible! The maximum betting stake x consistent with a lay stake of $20€$ can be derived from the equation $x * 3.0 - x = 20$, which results in $x = 10$. After matching these bets then the backing bet still has a remaining stake of $20 - 10 = 10€$, which can be matched against other lay bets.

In summary, backing an event means you bet on the event happening, while laying an event means you bet against the event happening. In a betting exchange platform, users can choose either option, allowing them to act as both bettors and bookmakers.

Matching

The matching algorithm is responsible for pairing back and lay bets from different users, ensuring that they have compatible odds and stake amounts. The main goal of the algorithm is to maximize the total matched amount of money while maintaining fairness and efficiency. Here's an overview of how a typical matching algorithm works:

1. The algorithm starts by collecting all pending back and lay bets for a specific market. These bets are usually stored in separate order books or queues, sorted by the odds and the time they were placed.

2. The algorithm compares the best available back and lay bets (i.e., the bets with the lowest back odds and the highest lay odds). If the back odds is lower than or equal to the lay odds, there is a potential match.

3. If a potential match is found, the algorithm calculates the matched amount by consuming either all of the backing bet stake, or all of the lay bet stake. That is, if the $backing\ stake * odds - backing\ stake \geq lay\ stake$ then all of the $lay\ stake$ is consumed. Otherwise all of the $backing\ stake$ is consumed.

4. After the matched amount is calculated, the algorithm updates the stake amounts of the back and lay bets. If either bet is fully matched, it's removed from the order book. If the bet is only partially matched, its stake is reduced by the matched amount, and the bet remains in the order book for future matching.

5. The matched bets, along with their odds and matched amounts, are recorded in the system. This information is used for tracking purposes and to update users' account balances after the market is settled.

6. The algorithm iterates through steps 2-5 until there are no more matches.

The matching algorithm typically operates in real-time or near-real-time, continuously processing new bets and updating the order books as users place, modify, or cancel their bets. This ensures that the platform remains responsive and up-to-date, providing users with an efficient and fair betting experience.

1.4 Example

Let's assume an event with the following unmatched bets:

	Back			**Lay**	
User	a	b	c	d	e
Odds	3	2	1.53	1.5	1.1
Stake	20€	14€	5€	21€	400€

The back and lay bets easiest to match are the bets from users c and d. User c risks 5€ expecting to win $2.65€ = 7.65 - 5$ ($7.65 = 5 \times 1.53$). User d is willing to risk 21€ expecting to win 42€ (solving the equation $x \times 1.5 - x = 21 \implies x \times 0.5 = 21 \implies x = \frac{21}{0.5} \implies x = 42$). The bets are close but do not match since the maximum odds that user d permits (1.5) is lower than the odds of user c (1.53).

If a user wants to make a new backing bet, its best option is the odds 1.5 offered by lay better d. If a user wants to make a new lay bet, its best option is the odds 1.53 by backing better c.

1. Let us assume that f backs the event with the odds 1.5 with a stake of 50€. The new state will be:

	Back				**Lay**
User	a	b	c	f	e
Odds	3	2	1.53	1.5	1.1
Stake	20€	14€	5€	8€	400€

   and

   - 42€ of the stake of f have been matched with the odds 1.5 against the user d ($21 = 42 \times 1.5 - 42$).

2. Next let us assume that g lays the event with a stake of 5€ with a (maximum) odds 1.53. The new state will be:

	Back			**Lay**
User	a	b	c	e
Odds	3	2	1.53	1.1
Stake	20€	14€	3.11€	400€

   and

   - 4€ of the stake of g have been matched with the odds 1.5 against the user f ($4 = 8 \times 1.5 - 8$), and
   - 1€ of the stake of g has been matched with the odds 1.53 against the user c ($1 = 1.89 \times 1.53 - 1.89$).

Note: In most betting exchange platforms, the lay stake represents the amount of money that the lay better is willing to cover from the perspective of a backing better. For example, the starting state in our example would be represented as:

	Back			**Lay**	
User	a	b	c	d	e
Odds	3	2	1.53	1.5	1.1
Stake	20€	14€	5€	42€	4000€

2 Project Objectives

The objective of this final project is to implement a scalable betting exchange system using Elixir, leveraging its fault-tolerant and concurrent features. The core functionality of the platform should be encapsulated in an Elixir module, which must include the basic functions required to place back and lay bets, manage user accounts, and match opposing bets.

2.1 Scalability Requirements

As the platform is expected to handle a large number of concurrent users and transactions, the implementation must be scalable. To achieve this, you should carefully consider the data structures (or databases) used, the algorithms for e.g. matching and so on. Moreover, it is advantageous to employ Elixir's OTP patterns, which will help in creating a robust, fault-tolerant, and scalable system. Proper use of OTP patterns, such as supervisors, GenServers, and Agents, is essential for meeting the project's scalability requirements.

Optional Tasks

If you desire there are a number of additional functionalities that your betting platform may provide. These are, for example, (i) a Telegram bot for betting,[4] (ii) a REST API for betting, and a (iii) Web application for betting.[5]

Note that such application normally authentificate users when betting. For this exercise, however, you are not required to implement user authentification (but you are allowed to do so should you so desire).

2.2 Project Deliverables

The following deliverables are expected for the successful completion of this final project:

1. A well-documented Elixir module implementing the core functionality of the betting exchange platform, relaying on OTP patterns for scalability.
2. A series of unit tests to validate the correctness of the implemented functions.
3. (Optional) A user-friendly Telegram bot interface, implemented using the ExGram library, for interacting with the betting exchange platform.
4. (Optional) A REST API for betting.
5. (Optional) A web interface for betting.
6. (Optional) Think and make a proposal to create a trading bot: an application that analyses the markets and try to find a bet that makes a user to always win (e.g, if the user backed an event 5 to 1 and now there is an unmatched back bet 3 to 1).

2.3 Required API

In this section we will describe the basic API that all solutions should implement. We will provide some basic tests that check that your API provides some basic level of functionality. The API should be implemented in a module named *BetUnfair*.

[4] The ExGram library (`https://github.com/rockneurotiko/ex_gram`) can be utilized, which provides an Elixir wrapper for the Telegram Bot API. This would allow users to interact with the platform through a familiar messaging interface.

[5] For implementing a REST API or a Web application the Phoenix framework (`https://www.phoenixframework.org/`) can be utilized to craft an API or an interactive Web app.

2.3.1 Odds and Stakes in API calls and return values

When stakes are used as arguments to API calls, or returned from API calls the amount of money represented as integers multiplied by 100, i.e. cents. Similarly returned account balances, and money deposits and withdrawals, are represented in cents. For example, the stake 2.25 is represented as 225, and the stake 11.333333 is represented as 1133 (*use truncation to convert the decimal number to an integer*). Similarly odds are represented as multiplied by 100 in API calls and values returned by the API (including as values in maps). For example, the odds 1.5 is represented in API calls as 150.

Note that internal calculations in the exchange which divides an amount by another amount should be done using integer division (i.e., truncation).

2.3.2 Persistence

Your exchange should be persistent. That is, a library for storing information persistently (on disk) such as *dets, CubDB* (`https://github.com/lucaong/cubdb`), or a normal database behind *Ecto* (`https://github.com/elixir-ecto/ecto`), can be used to store exchange data.

2.3.3 Function specification

The expected type of function arguments, and the value returned by a function, will be indicated using the normal Elixir function specification style, documented in `https://hexdocs.pm/elixir/main/typespecs.html#defining-a-specification`. For example,

```
@spec user_withdraw(id :: user_id(), amount :: integer()) :: :ok | :error
```

specifies that first argument of the `user_withdraw` function is expected to be a string, and the second argument an integer, and it returns either the atom `:ok` or the atom `:error`.

2.3.4 Return Values

All functions signal whether their execution was successful or not. A successfully execution is indicated by the atom `:ok` or a tuple `{:ok, result}`. In the following if a function is specified to return an atom `:ok` you *may* instead return a tuple `{:ok,result}`. In case a function is specified to return a tuple where the result is a map, e.g., `{:ok, %{id: user_id(), balance: integer()}}` the tuple returned *must* contain the specified keys (e.g., `id` and `balance`) but *may* contain additional keys.

An unsuccessful result is signaled if the function returns either the atom `:error` or tuple `{:error, reason}`. In the following function specifications **only the successful result type will be specified**.

2.3.5 Returning a large number of elements

Some functions potentially return a large number of elements, e.g., `market_bets` should return all bets in a market which, when your exchange becomes a commercial success `:-)`, may be a lot of bets. Such functions will be specified, in a non-standard manner, with the return type *`Enumerable`*`.t(bet_id())` (where `bet_id()` is the type of elements returned by `market_bets`; other functions return other types of elements). You are *encouraged* to implement such enumerables as lazy streams (see `https://hexdocs.pm/elixir/1.14/Stream.html`) which can yield a better performance if not all returned bets needs to be inspected. Alternatively, you can for instance use ordinary lists.

2.3.6 Required Functions

Your *`BetUnfair`* module *must* implement the following functions:

Exchange interaction

- `start_link(name :: string()):: {:ok, _}` – starts the exchange. If an exchange name does not already exist it is created. If it exists the existing data (markets, bets, users) is recovered.
- `stop():: :ok` – shuts down a running exchange in an orderly fashion (preserving exchange data).
- `clean(name :: string()):: :ok` – stops the exchange if it is running, and removes all persistent data (from disk).

User interaction

- `user_create(id :: string(), name :: string()) :: {:ok, user_id()}`

  adds a user to the exchange. The id parameter must be unique (e.g., a DNI/passport number, email, username). Fails, for instance, if a user with the same id already exists. Returns a user identifier (you may choose its representation).
- `user_deposit(id :: user_id(), amount :: integer()):: :ok` – adds amount (should be positive) to the user account.
- `user_withdraw(id :: user_id(), amount :: integer()):: :ok` – withdraws amount (should be positive, and the account balance must be at least amount) to the user account.
- ```
  user_get(id :: user_id()) ::
    {:ok, %{name: string(), id: user_id(), balance: integer()}}
  ```

  retrieves information about a user.
- `user_bets(id :: user_id()):: Enumerable.t(bet_id())` – returns an *enumerable* containing all bets of the user.

Market interaction

- ```
  market_create(name :: string(), description :: string()) ::
    {:ok, market_id()}
  ```

  creates a market with the unique name, and a potentially longer description.
- `market_list():: {:ok, [market_id()]}` returns all markets.
- `market_list_active():: {:ok, [market_id()]}` returns all active markets.
- `market_cancel(id :: market_id()):: :ok` – cancels a non-terminated market. Returns all stakes in matched or unmatched market bets to users.
- `market_freeze(id :: market_id()):: :ok` – stops all betting in the market; stakes in non-matched bets are returned to users.
- `market_settle(id :: market_id(), result :: boolean()) :: :ok`

  the market (event) has been resolved with argument result. Winnings are distributed to winning users according to stakes and odds.
- `market_bets(id :: market_id()) :: {:ok, Enumerable.t(bet_id())}`

  all bets (matched or unmatched) for the market are returned.

- `market_pending_backs(id :: market_id()) ::`
  `{:ok, Enumerable.t({integer(), bet_id()})}`

  all pending (non matched) backing bets for the market are returned. **Note** that the bets are returned as a tuple `{odds,bet_id}`, and that the elements should be returned in *ascending* order (i.e., bets with smaller odds first).

- `market_pending_lays(id :: market_id()) ::`
  `{:ok, Enumerable.t({integer(), bet_id()})}`

  all pending lay bets for the market are returned. **Note** that the bets are returned as a tuple `{odds,bet_id}`, and that the elements should be returned in *descending* order (i.e., bets with larger odds first).

- ```
  market_get(id :: market_id()()) ::
    {:ok, %{name: string(),
            description: string(),
            status: :active |
                    :frozen |
                    :cancelled |
                    {:settled, result::bool()}}}
  ```

- `market_match(id :: market_id()):: :ok` – try to match backing and lay bets in the specified market.

Bet interaction

- ```
  bet_back(user_id :: user_id(),
           market_id :: market_id(),
           stake :: integer(),
           odds :: integer()) :: {:ok, bet_id()}
  ```

  creates a backing bet by the specified user and for the market specified.

- ```
  bet_lay(user_id :: user_id(),
          market_id :: market_id(),
          stake :: integer(),
          odds :: integer()) :: {:ok, bet_id()}
  ```

  creates a lay bet by the specified user and for the market specified.

- `bet_cancel(id :: bet_id()):: :ok` – cancels the parts of a bet that has not been matched yet.

- ```
  bet_get(id :: bet_id()) ::
    {:ok, %{bet_type: :back | :lay,
            market_id: market_id(),
            user_id: user_id(),
            odds: integer(),
            original_stake: integer(),  # original stake
            remaining_stake: integer(), # non-matched stake
            matched_bets: [bet_id()],   # list of matched bets
            status: :active |
                    :cancelled |
                    :market_cancelled |
                    {:market_settled, boolean()}}}
  ```