Task Description

Implement a REST API for a leaderboard service. API endpoints and what the client expects to receive from them are defined below.

The leaderboard service groups players to competitions of 10 players, balancing the matchmaking waiting time and grouping participants with others that are close to their level (and as a bonus, if you have time, country). Both data points for matchmaking are given in a JSON document described below. Once enough players have joined a competition (or 30 seconds of waiting has passed), the competition starts. The competition runs for 60 minutes, during which players can submit points. The points are added incrementally to players' total points in this competition.

The REST API allows retrieving the player's current competition by player ID, or any past or current competition by competition ID. In both endpoints, the player IDs and their total score are returned, ordered by the points. When two players are tied, they are sorted alphabetically by player ID.
After the competition has ended, the player can no longer submit points to it and needs to join another leaderboard competition to compete again. In the new competion, the player starts from zero points.

API Endpoints

`POST /leaderboard/join?player_id=<string ID>`

A player calls this endpoint to start matchmaking into a leaderboard competition.
If a player has already joined a competition and that competition has not ended, they cannot join a new one until the current competition ends.
Success responses:
If a matching competition is found right away:

```
{
  "leaderboard_id": "<string uuid>",
  "ends_at": <timestamp>,
}
```

If a competition fitting the player is not found right away, the player needs to wait in a matchmaking queue, and the service returns `HTTP 202 (Accepted)`.
If the player is already in an active competition and cannot join, the service returns `HTTP 409 (conflict)`.
When picking a player's competition group, you can use the following player data structure (assuming it's coming from a database, but you don't need to write the code
for retrieving it):

```
{
  "level": <int>,
  "country_code": "<string>",

}
```

Note: keep concurrency in mind, e.g. what if two players join a competition at the same time?
Also, matchmaking shouldn't take more than 30 seconds. So, when designing your matchmaking algorithm, balance the time it takes to find the match with how close the players are to each others' levels, in the end, placing the player in the closest matching group.
If you have time, it would be nice to see a similar implementation based on country code, but the requirement is to just handle player level as input data.

`GET /leaderboard/player/<player_id>`

The player can only participate in one leaderboard at a time, so this endpoint returns the information about the player's current leaderboard, or the last one they joined, if the current one has finished.
This can be used e.g. to check if the matchmaking was completed.
Returns:

```
{
  "leaderboard_id": "<string id>",
  "ends_at": <timestamp>,
  "leaderboard": [
    {
      "player_id": "<string id>",
      "score": <int>
}, {
      "player_id": "<string id>",
      "score": <int>
    },
... ]
}
```

If the player is not in a leaderboard, return an empty JSON object.

`GET /leaderboard/<leaderboardID>`

Returns the information about a specific leaderboard competition. This can be useful for e.g. returning historical leaderboard information.
Returns:

```
{
  "leaderboard_id": "<string id>",
  "ends_at": <timestamp>,
  "leaderboard": [
    {
```

```
      "player_id": "<string id>",
      "score": <int>
}, {
      "player_id": "<string id>",
      "score": <int>
    },
... ]
}
```

If a leaderboard with the given ID doesn't exist, returns `HTTP 404 (Not found)`.

`POST /leaderboard/score`

Submits a score to the player's current leaderboard. The submitted score is added to the player's total score without server side validation.
JSON body:

```
{
  "player_id": "<string id>",
  "score": <int>
}
```

If the player has not joined a competition or the latest competition is over, the service returns `HTTP 409 (conflict)`.

In a success case, the service returns HTTP 200 (OK) without a body.

Bonus Points

Country-aware grouping in matchmaking.

Context propagation (context.Context).

Graceful shutdown via http.Server.Shutdown.

Prometheus metrics (client_golang).

Static analysis (go vet, staticcheck, golangci-lint).

CI – GitHub Actions running go test ./....

Deliverables

1. Git repository containing:

- /cmd/server – entry-point main.go.
- /internal or /pkg with domain logic.
- go.mod, go.sum, Dockerfile, docker-compose.yml.

2. README.md covering:
    - Setup (docker compose up) & non-Docker run instructions.
    - Design decisions & trade-offs.
3. Automated tests – go test ./... passes.
4. (Optional) short Loom / markdown architecture walk-through.

Evaluation Criteria

Correctness vs. spec, especially race-free matchmaking and score updates.

Test coverage & clarity.

Docker + Compose experience – project comes up cleanly on Unix.

Extra polish from any bonus features.