Backend developer assignment (Elixir)

Introduction

First of all, congratulations on getting to this stage of the interview process. This is the stage where you get to show off your technical abilities.

Take your time and read the instructions carefully. The assignment is pretty open-ended to give you space to present your software architecture and creativity skills. A lot of details have been intentionally left out for just this reason. However, if you feel like you are stuck because of unclear instructions you can send us a message and we will clear it up.

Give it your best shot and submit a project that you are proud of.

What you will be building

The task is to build a barebones Massively multiplayer online game (MMO). The game is very simple in terms of graphics (frontend) and gameplay mechanics. The main point is to demonstrate your skills and understanding of Elixir, the OTP and the basics of web development.

The candidate should ideally solve this assignment in 8 hours. If not completed in time, don't give up! Do your best and submit your assignment as soon as possible.

What the game should look like

The game is played on a 2D grid. The player is looking from the top down. Each field in the grid can be either a **wall** or a **walkable/empty** tile. The “map” can be predefined (hardcoded) and does not need to be randomly generated. The only condition is that there must be at least some wall tiles scattered throughout the grid. Check the attached examples for inspiration. Feel free to copy the design of the provided example grid or design your own.

The player controls a hero, which is rendered on the grid. The hero can move freely on the empty tiles, but **not** on the wall tiles. When other players connect to the game, your enemies, they are also rendered on the grid. Your hero should always be distinguishable from the enemies.

While moving, if an enemy is already on a tile, your hero can still move to that tile. Your hero should be rendered above the enemies so that your hero is always visible. When your hero or an enemy is dead, it should be distinguishable from others. Again, consult the attached examples for ideas.

The game mechanics

- The game is played in the browser
- The player is assigned a hero which is spawned on a random walkable tile. The hero is assigned a random name. The player can also choose a custom name for their hero
- If a player connects to the game with a name that already exists and is controlled by another player, both players control the same hero. You don’t need to make your game cheat-proof in this regard
- Your hero can move freely over all empty/walkable tiles. They can also walk on tiles where enemies are already standing
- Each hero can attack everyone else within the radius of 1 tile around him (in all directions) + the tile they are standing on. For example, if your hero is standing on tile {2, 3}, they can attack everyone on tiles: {1, 2}, {2, 2}, {3, 2}, {1, 3}, {2, 3}, {3, 3}, {1, 4}, {2, 4}, {3, 4}.
- If there are multiple enemies in range, all of them are attacked at the same time. One hit is enough to kill the enemy.
- If an enemy attacks you, your hero dies. When your hero is dead, it cannot perform any actions
- 5 seconds after dying, the hero re-spawns at a random location

Implementation rules

- Each hero is represented by a **GenServer** which holds their position on the grid and the status (alive/dead)
- The game interaction does not have to be “live” (soft-real time). There is no need to use WebSockets. You may use them if you wish, but it’s not a requirement. The graphics (frontend) update loop may be as simple as a page refresh every 1 second.
- We suggest creating a route such as `/game` to play the game
- We recommend letting players choose the name of their hero with query parameters, such as `/game?name=Geralt`
- The player input can either be from a keyboard or as simple as some HTML buttons

What your focus should be on

- Software architecture and clean code. Take your time. Think about separation of concerns and how to make the code readable and testable
- Production-ready code. Clear your debug messages, TODO comments, and unneeded behaviors. Imagine you are shipping a piece of software to production
- Utilizing the OTP. We want to see how you use GenServers, Supervisors, and message passing in general to your advantage
- At least some unit tests where you test your business logic
- At least one GenServer test

What your focus should NOT be on

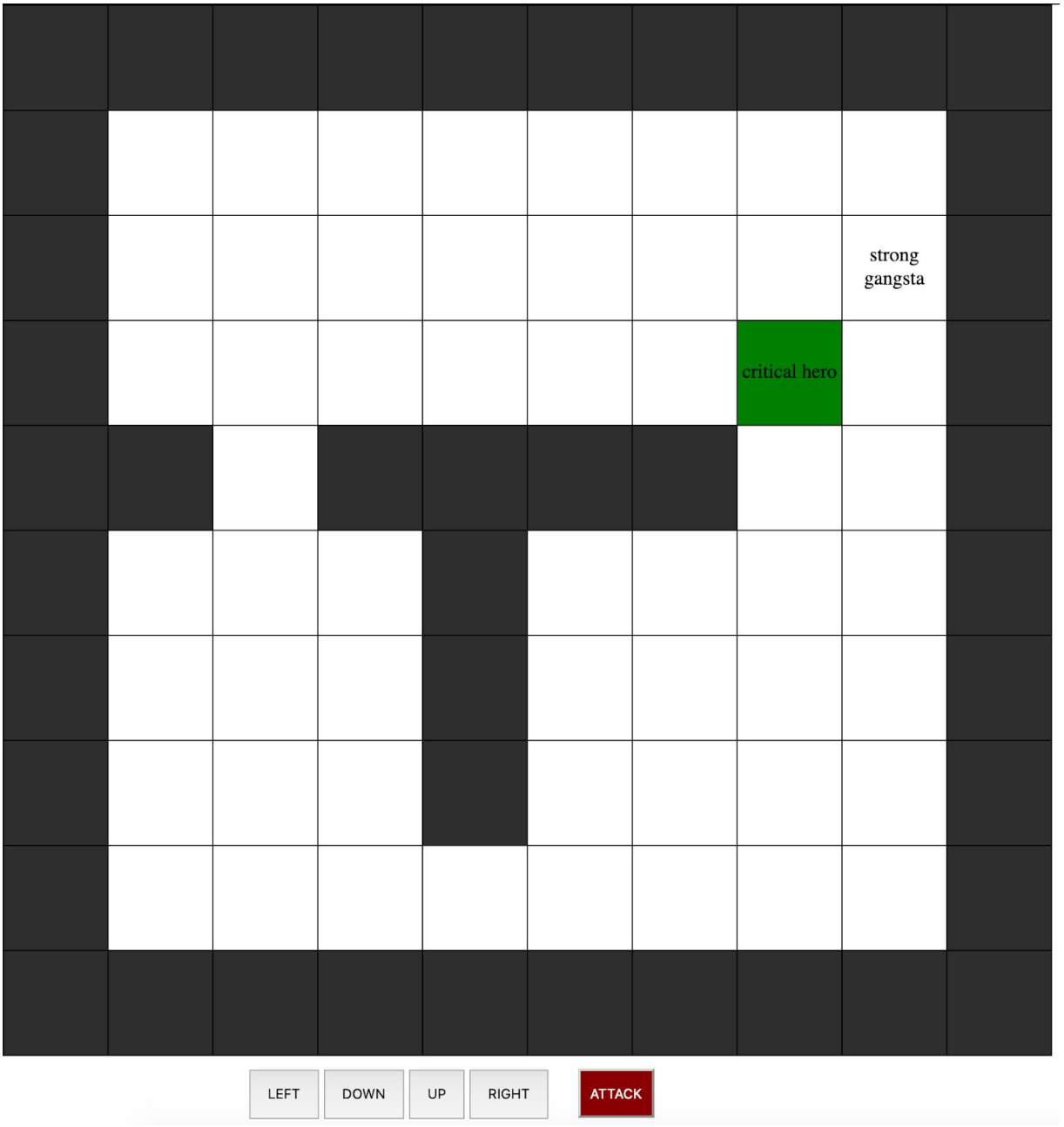
- Frontend code. There is no need for fancy frontend frameworks and libraries. Make it as basic as possible. Check out the attached examples for some inspiration
- The graphics/game assets and interactions. Again, focus on the software architecture, not on the looks and interaction
- Libraries. The focus is to use as much vanilla Elixir as possible. The assignment is set in such a way that you don’t really need to use many libraries. One library we do recommend using is Phoenix for the web part of your codebase

What you should deliver

- Link to a git repository where the project resides (we recommend GitHub)
- Instructions (Readme.md) on how to **run the project in the dev environment**, **how to run tests** and **how to build the production release** (we recommend using Elixir releases)
- Link to a working release. We recommend gigalixir or Heroku

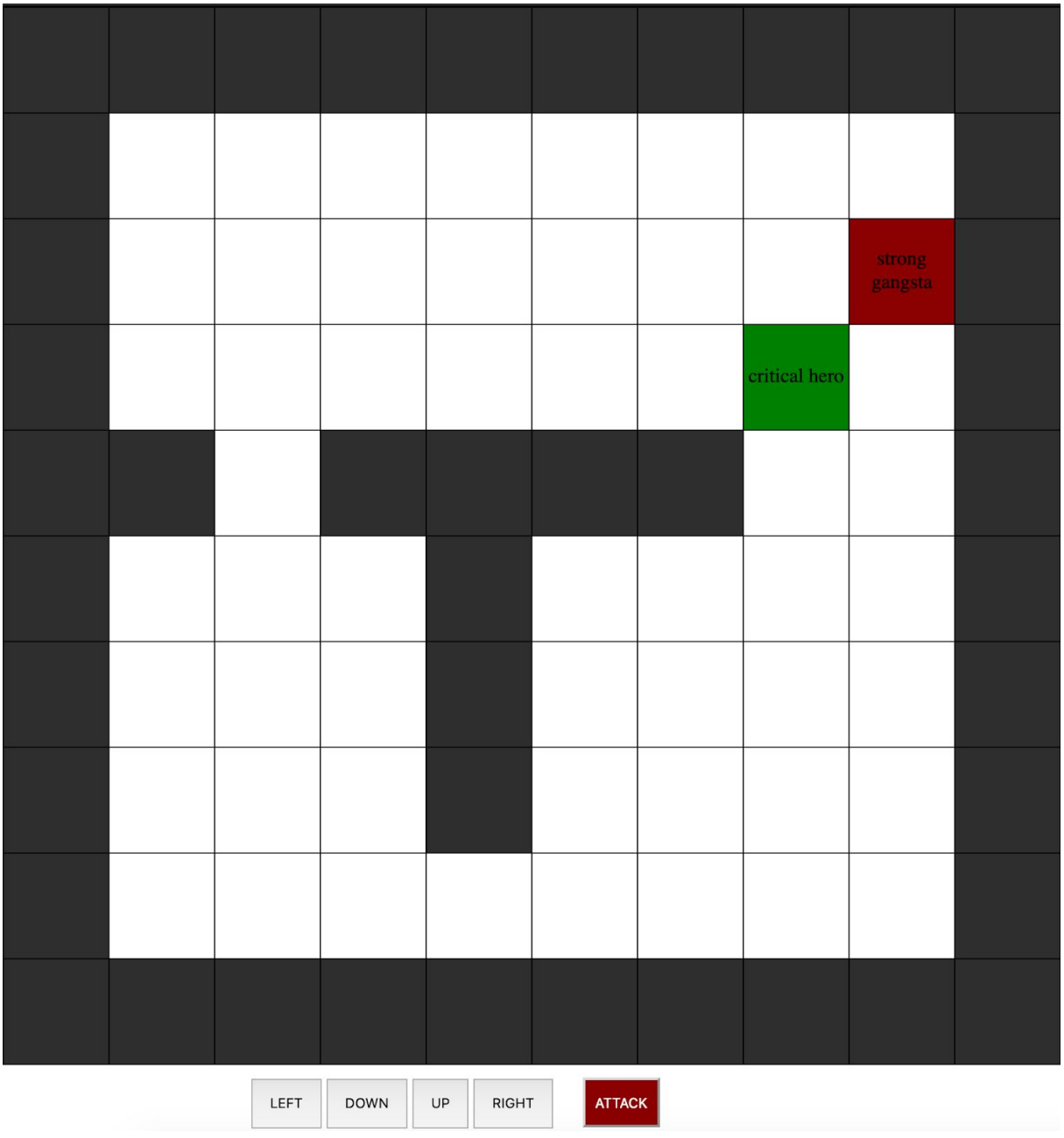
Example #1

Notice how there is minimal work being done on the frontend. All we did was make a simple grid with a few buttons to control the character. The wall tiles are a dark gray and the empty tiles are white. Your hero is distinguishable from other tiles. In our case, we made the hero green.



Example #2

When an enemy is within 1-block radius (or in the same position as your hero), triggering the “attack” command kills the enemy player.



Example #3

Alternatively, when your hero is dead you cannot perform any actions. You can only spectate the game. 5 seconds after dying, the hero can be controlled again.

Your hero is dead