CHECKERS

by
Marcin Szewczyk

GitHub github.com/marcinszewczykArch/checkersOne

HEROKU checkersTwo.herokuapp.com

GOALS

- Implement **checkers game** model and rules based on brasilian checkers
- Create both **single player** game (play against computer) and **multi player** game
- Create simple and intuitive **frontend**
- Touch as **many topics** from Scala Bootcamp as possible
- Have **fun** ;)

PROJECT COMPONENTS

$$\begin{tikzcd}\text{Scala} \arrow[r, leftrightarrow, "\text{websocket}", "\text{http}"'] & \text{ANGULAR CLI}\end{tikzcd}$$

Backend

- Checkers domain and move validation
- Single player game (http):
    - simple AI
- Multi player game (websocket):
    - player creation
    - room creation
    - handling game
    - chat
- DB connection:
    - save game state
    - load game state

Frontend

- Main page :
    - choose game type
    - choose colour
    - choose initial state
- Multi player login page with validation
- Rooms page with multi player game state info
- Game page wiith:
    - board
    - game state info
    - chat
- static pages:
    - game rules
    - about

CI/CD AND DEPLOYMENT

GitHub Actions For Backend (Running tests), Deployment to **Heroku** if the workflow succeeds

GitHub Actions

heroku

LIBRARIES

cats

cats-effect

http4s

fs2

circe

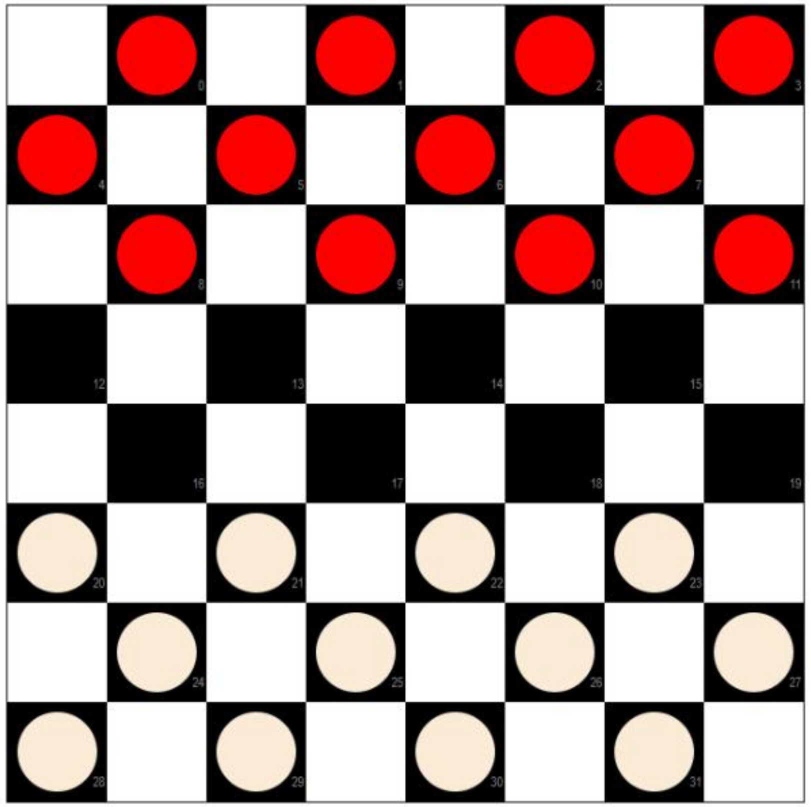
doobie

+ enumeratum
+ scalatest
+ scalafmt

GAME RULES

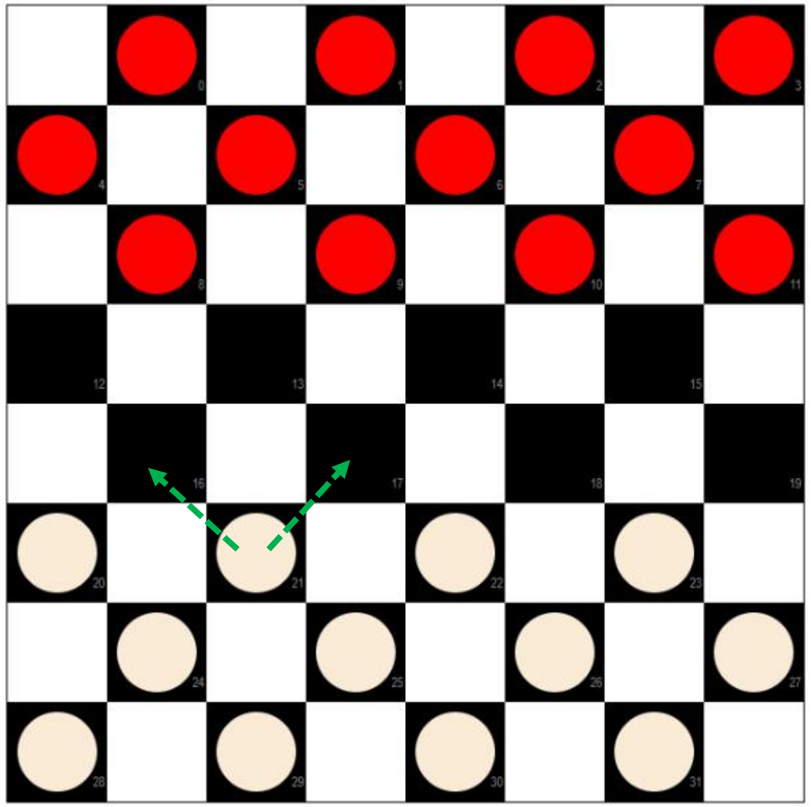
RULES

1. **Initial game state** looks like shown below. The player with **white** pieces makes the **first move.**



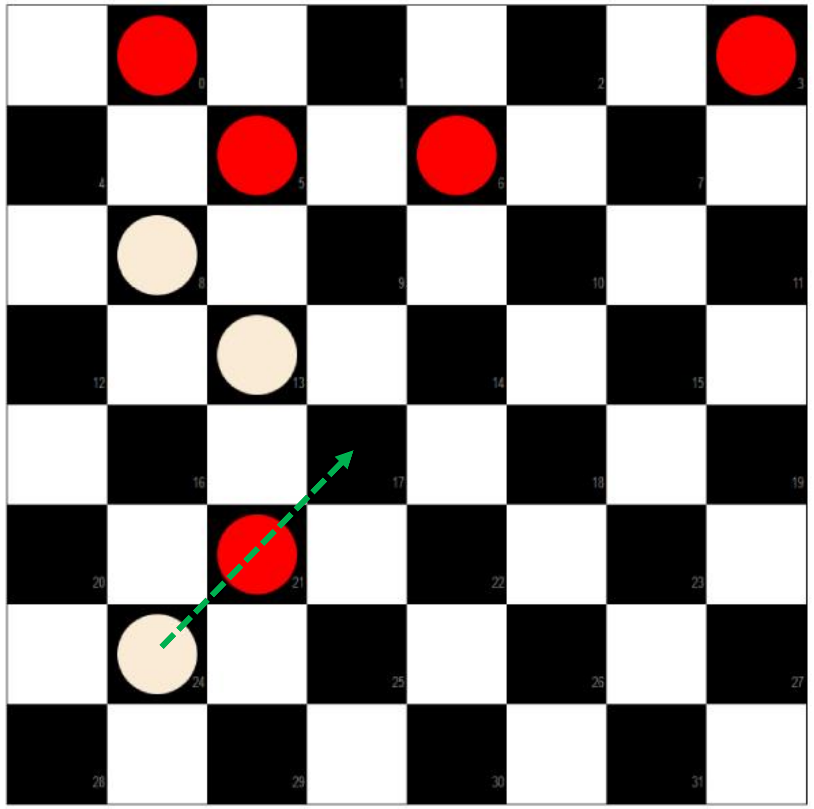
RULES

2. Regular pieces move **forward** one square **diagonally** to a square that is not occupied by another piece.



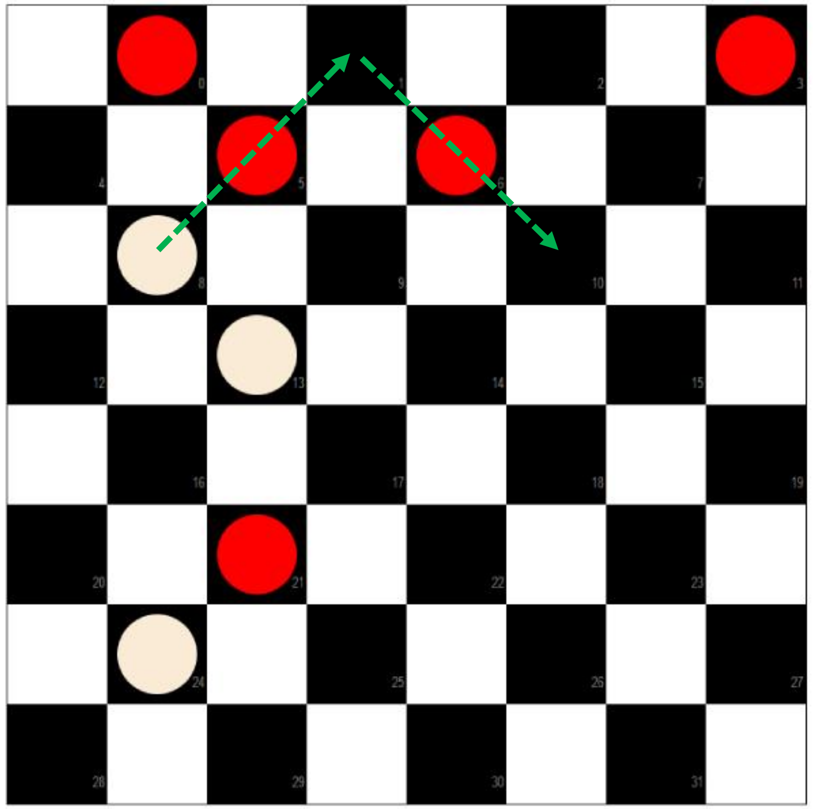
RULES

3. Opposing pieces can and must be captured by jumping over the opposing piece (**diagonally backward** or **forward**). Capturing a piece is **obligatory**.



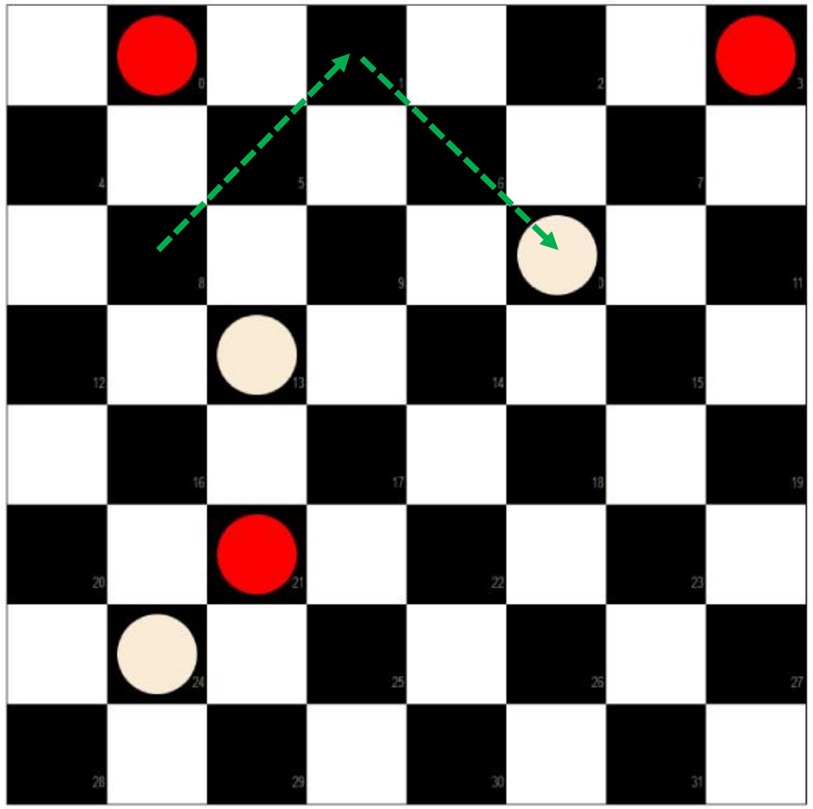
RULES

4. If there is **next piece to capture** by **the same piece** it has to be done in the same round (**multiple capturing**).



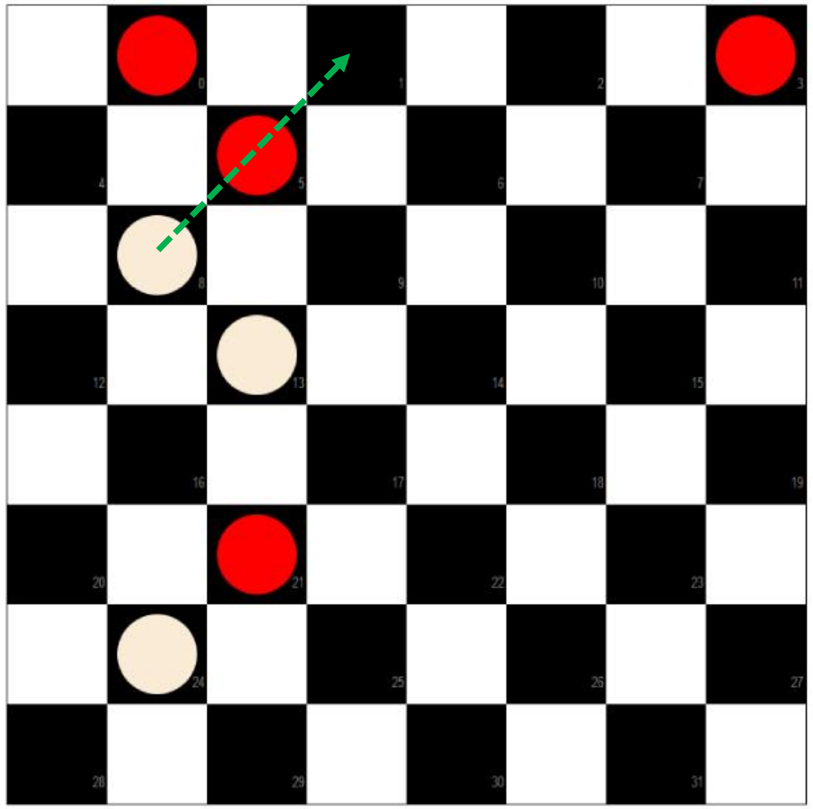
RULES

5. After the piece has jumped over the opponent's piece or pieces, the jumped-over pieces are **taken from the board**.



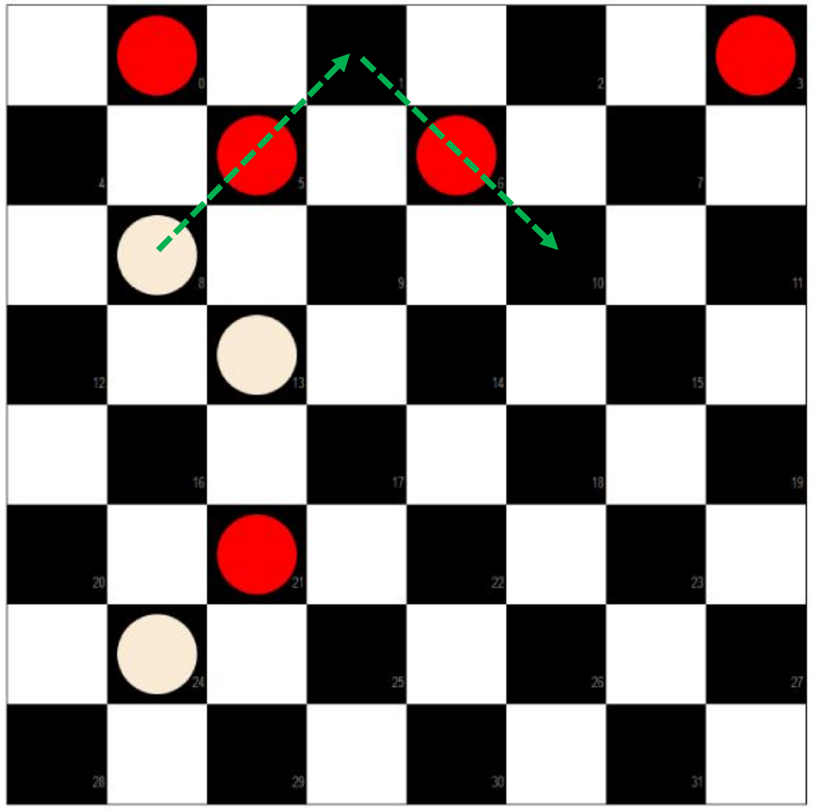
RULES

6. A piece **becomes** a **queen** if it stops on the far edge of the board at the end of its turn.



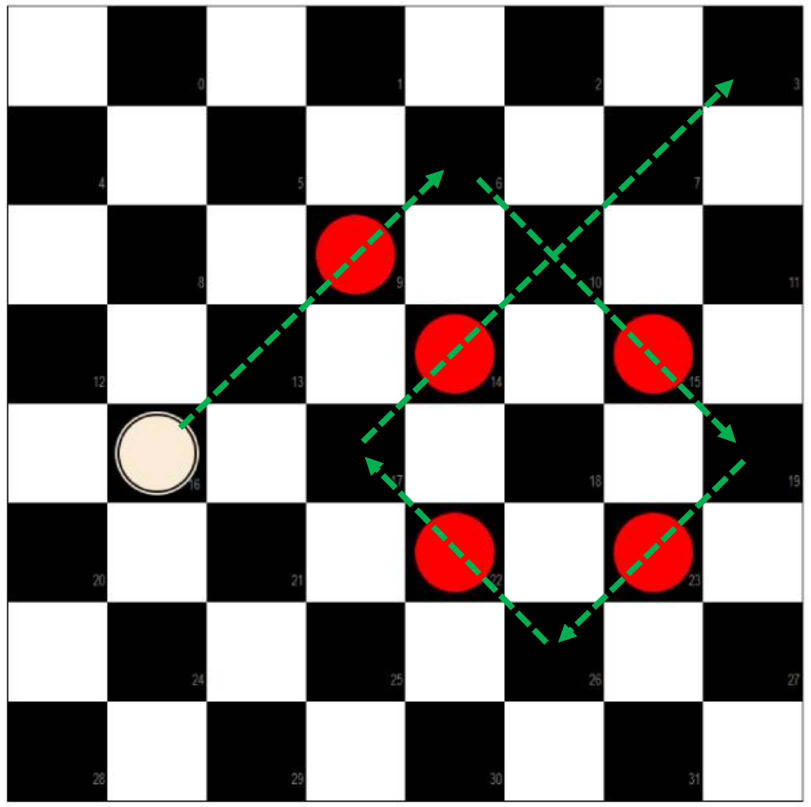
RULES

7. A piece does **not become** a **queen** if it reaches the edge but must then jump another piece backward.



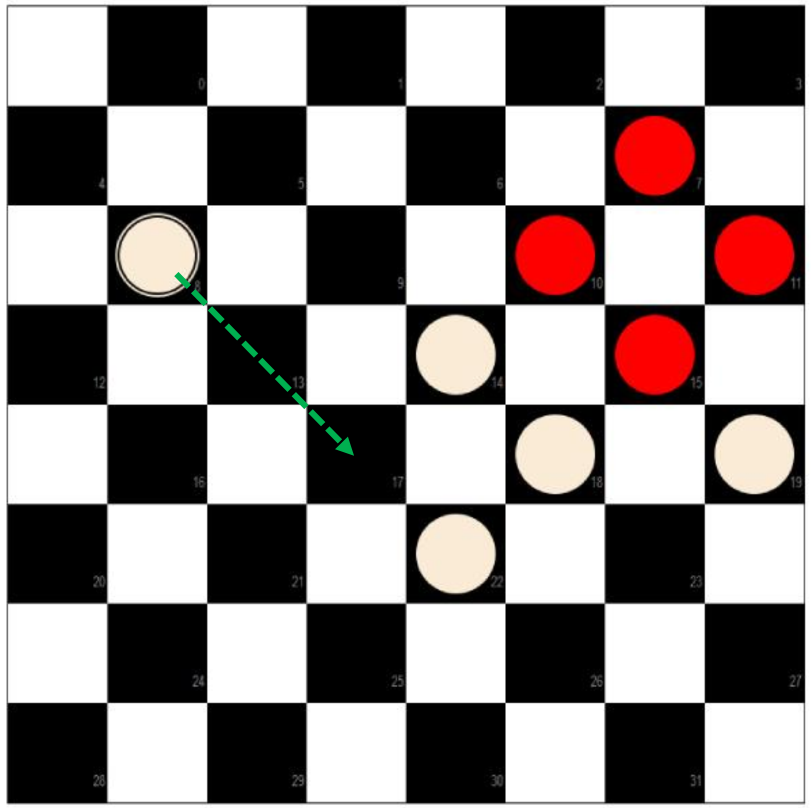
RULES

8. **Queen** pieces can move freely **multiple steps** in **any direction** and may jump over and hence capture an opponent piece some distance away and choose where to stop afterwards.



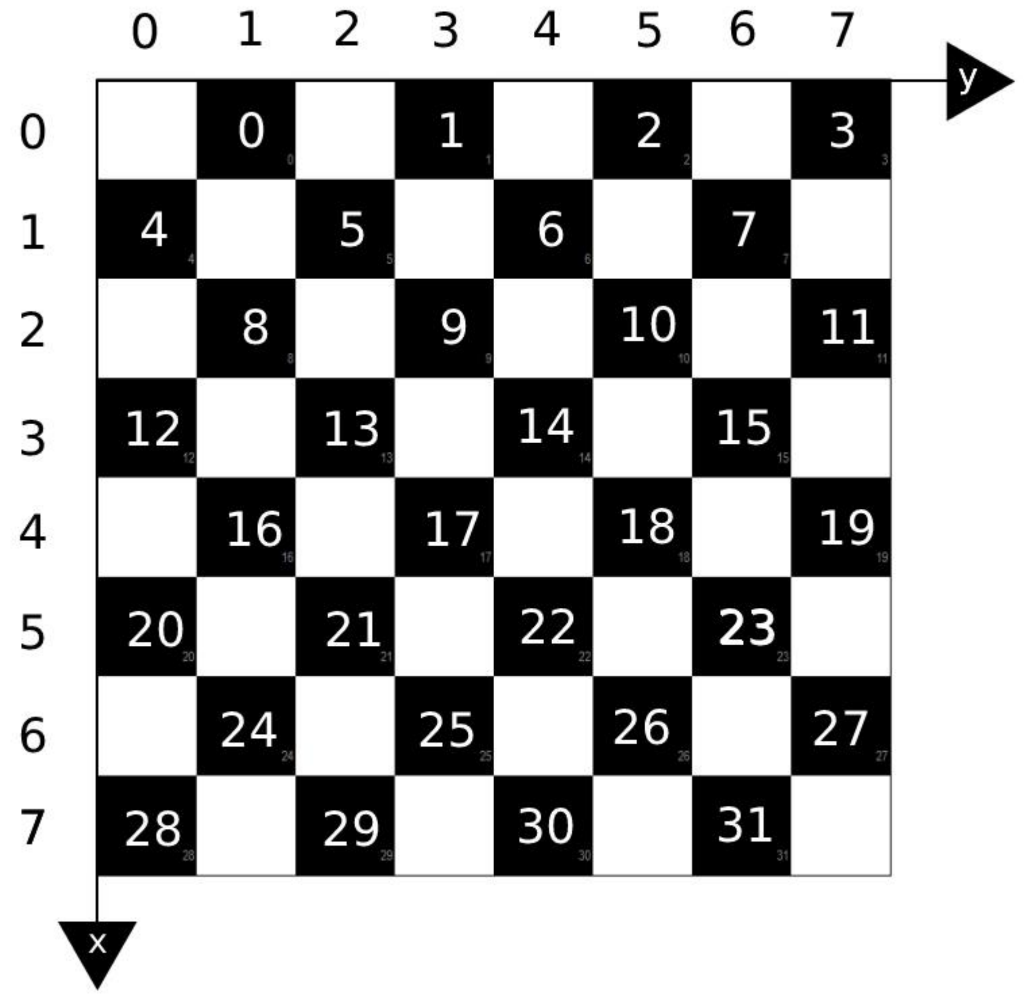
RULES

9. A player with **no valid move** remaining loses. This is the case if the player either has **no pieces left** or if a player's pieces are **obstructed from making a legal move** by the pieces of the opponent.



NOTATION

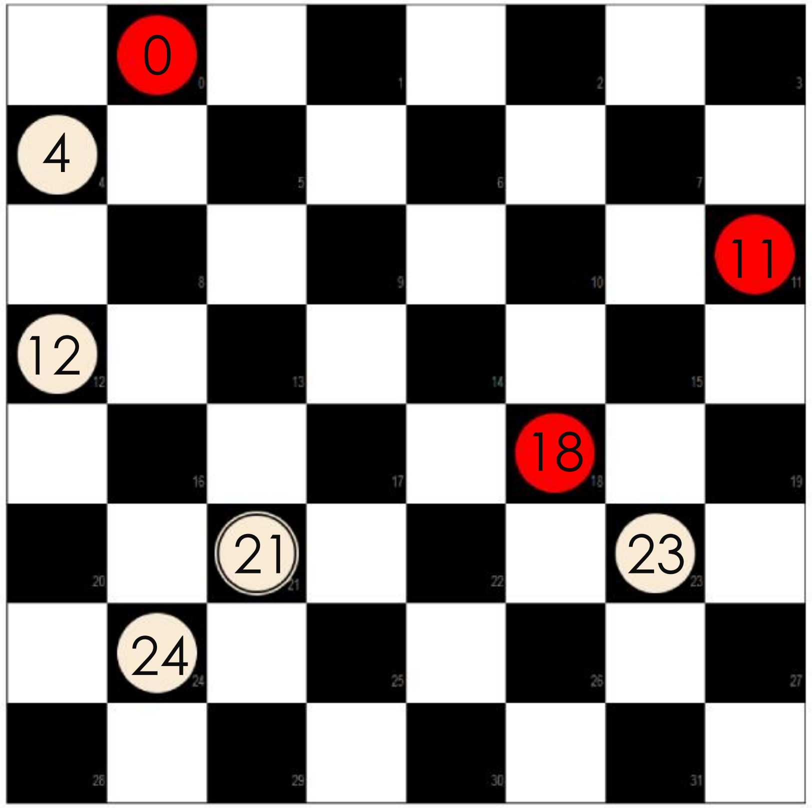
Each of the **32 dark squares** has a number (***0* through *31***). Number 28 is at the left corner seen from the player with the white pieces. Number 3 is at the left corner seen from the player with the red pieces.



BOARD AS STRING

Board string is interpreter as **Map[index, char]** where single char represents **pawn type** and **pawn colour**

r	o	o	o	w	o	o	o	o	o	o	r	w	o	o	o	o	o	r	o	o	W	o	w	w	o	o	o	o	o	o	o
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

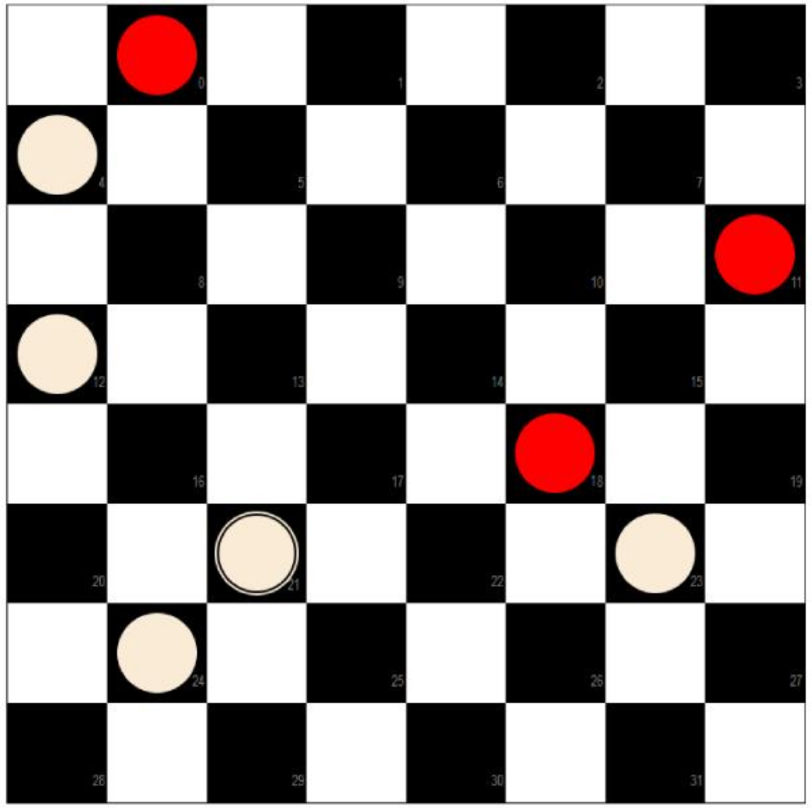


<u>Char to Pawn</u>

o	– empty
r	– regular, red
w	– regular, white
R	– queen, red
W	– queen, white

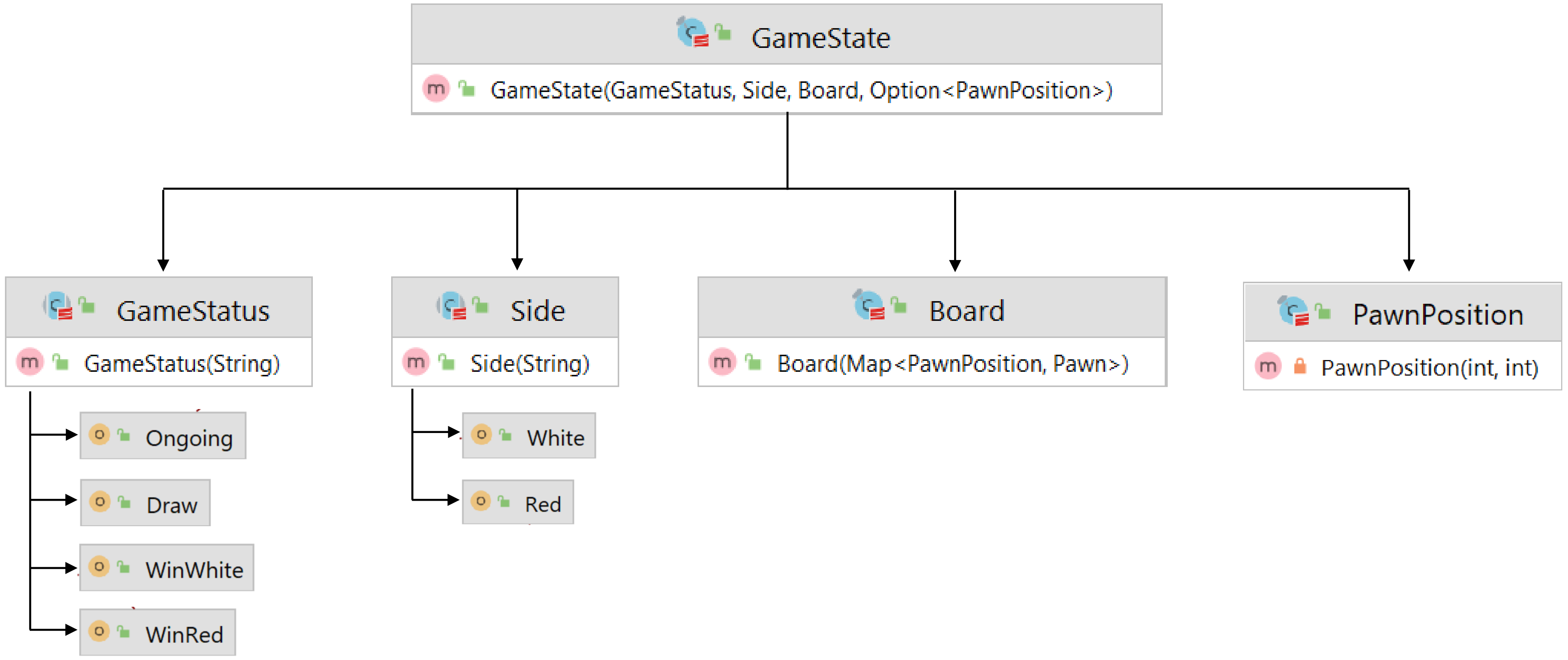
GAME STATE AS JSON

```
{
   "status": "ongoing",
   "movesNow": „w",
   "board": "rooowoooooorwooooorooWowwooooooo",
   "nextMoveFrom": 23
}
```

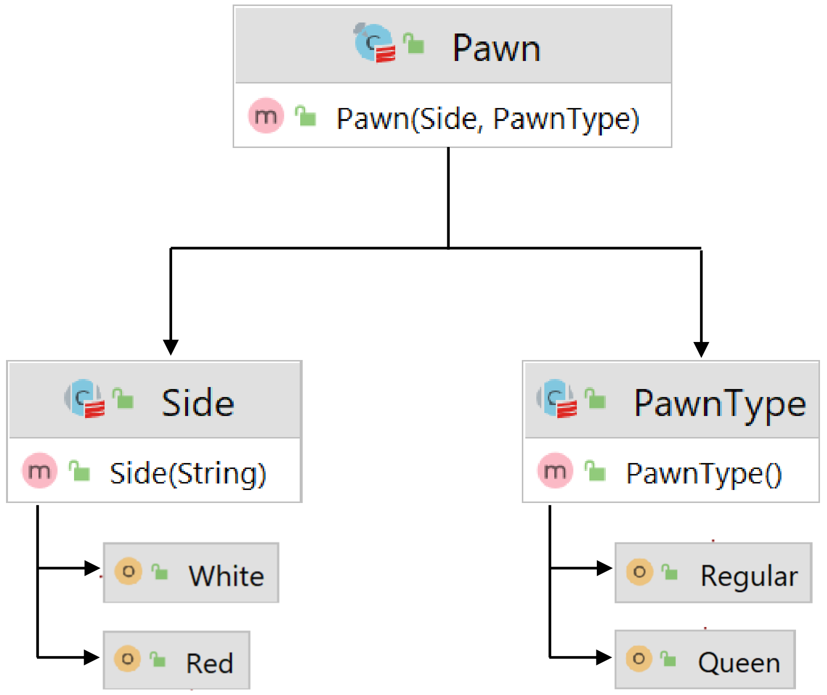


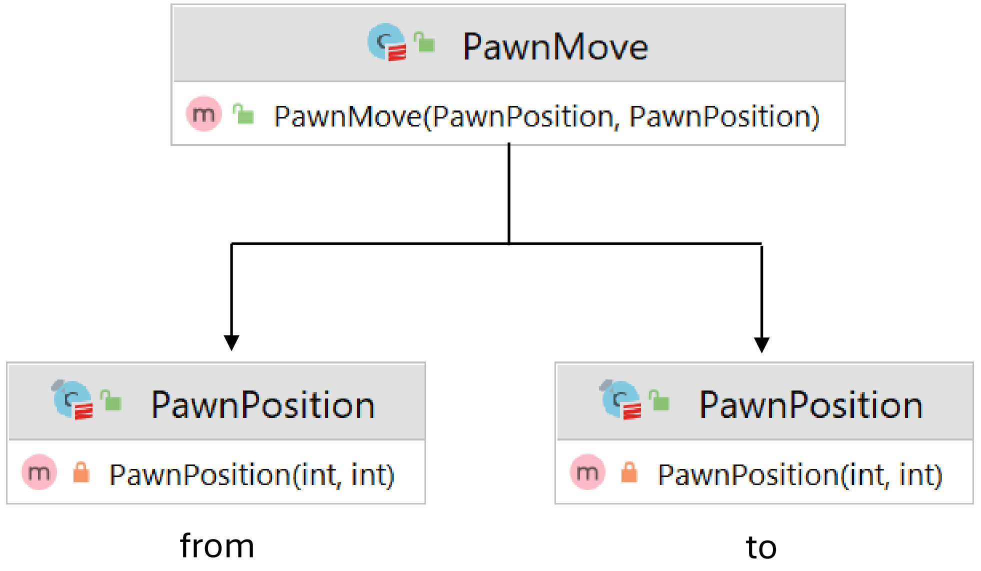
GAME STATE

```
{
    "status": "ongoing",
    "movesNow": „w",
    "board": "rooowoooooorwoooooroooWowwoooooooo",
    "nextMoveFrom": 23
}
```

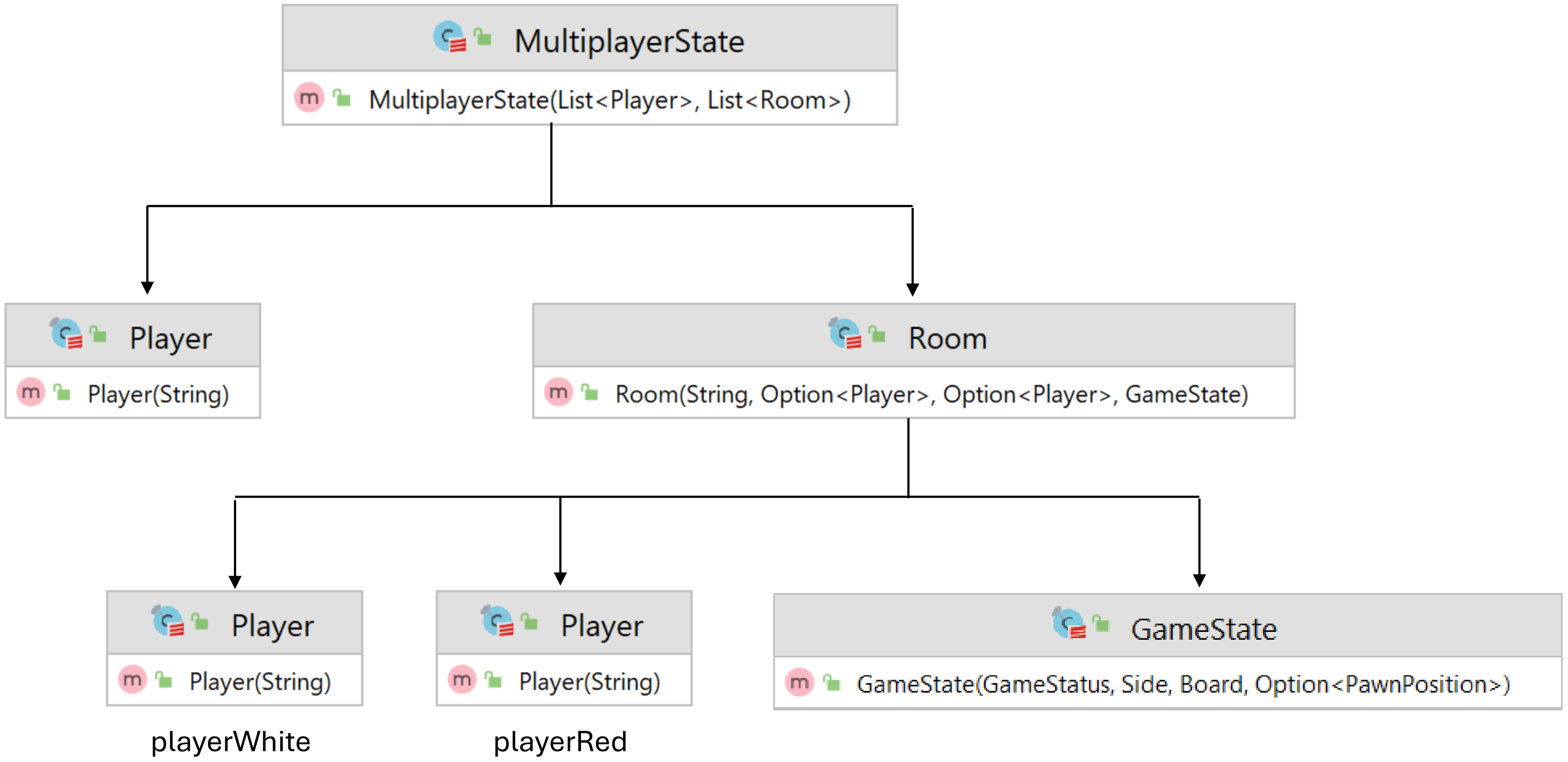


PAWN and PAWN MOVE

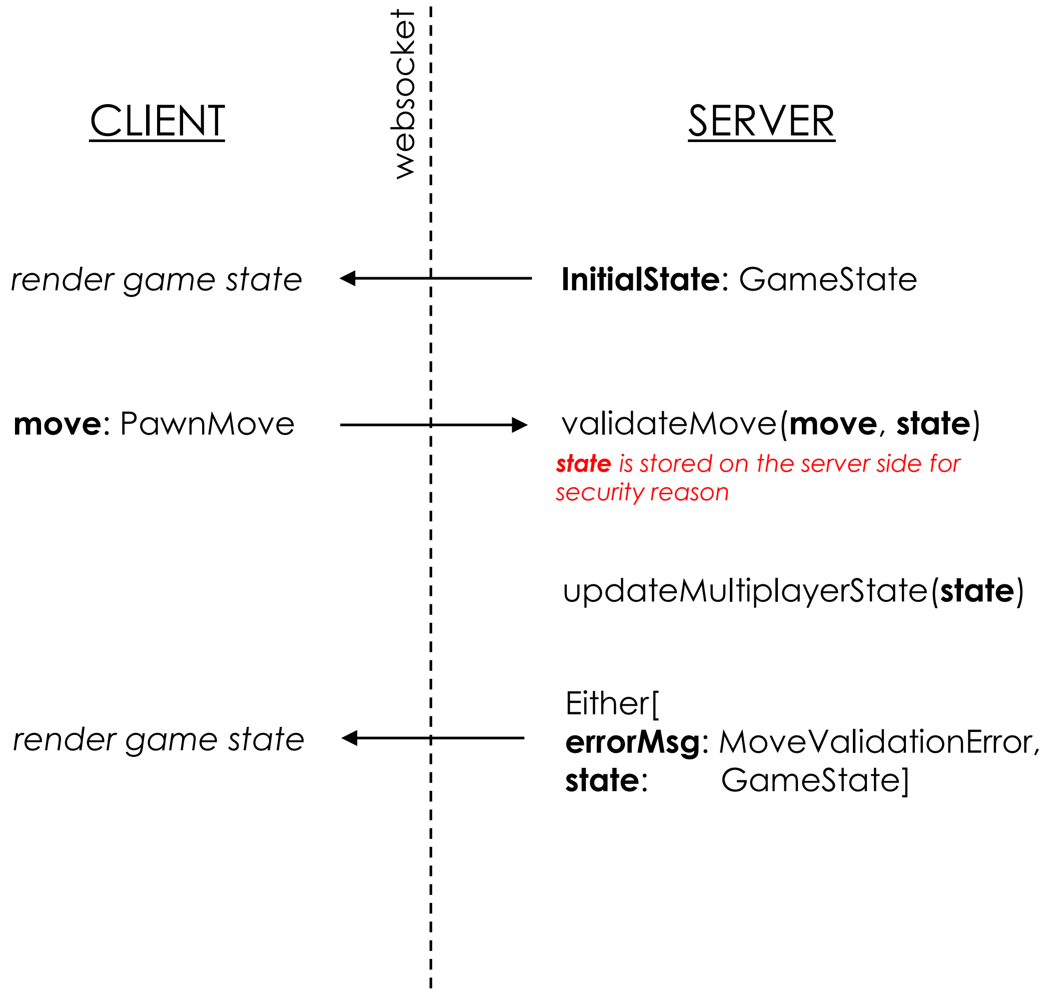




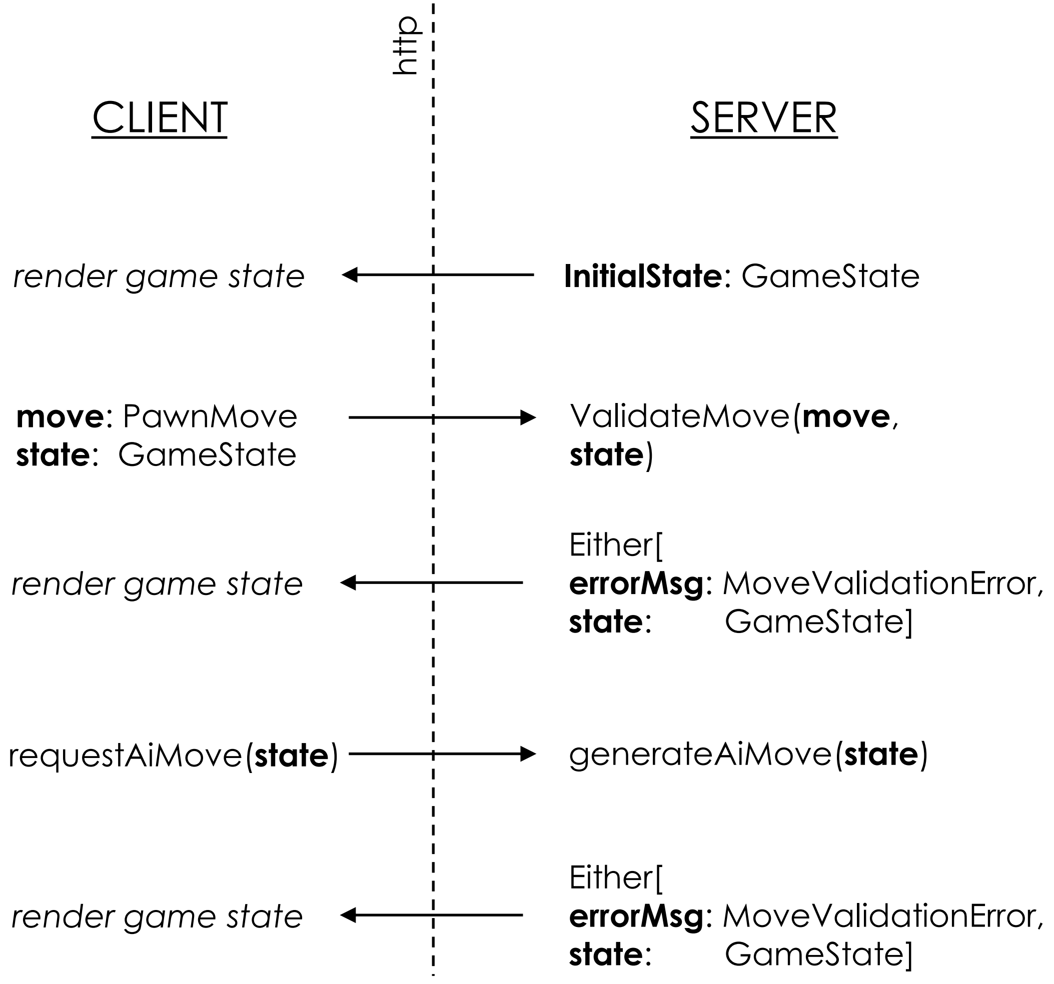
MULTIPLAYER STATE


MultiplayerState
MultiplayerState(List<Player>, List<Room>)
Player
Player(String)
Room
Room(String, Option<Player>, Option<Player>, GameState)
Player
Player(String)
playerWhite
Player
Player(String)
playerRed
GameState
GameState(GameStatus, Side, Board, Option<PawnPosition>)

CLIENT-SERVER COMMUNICATION

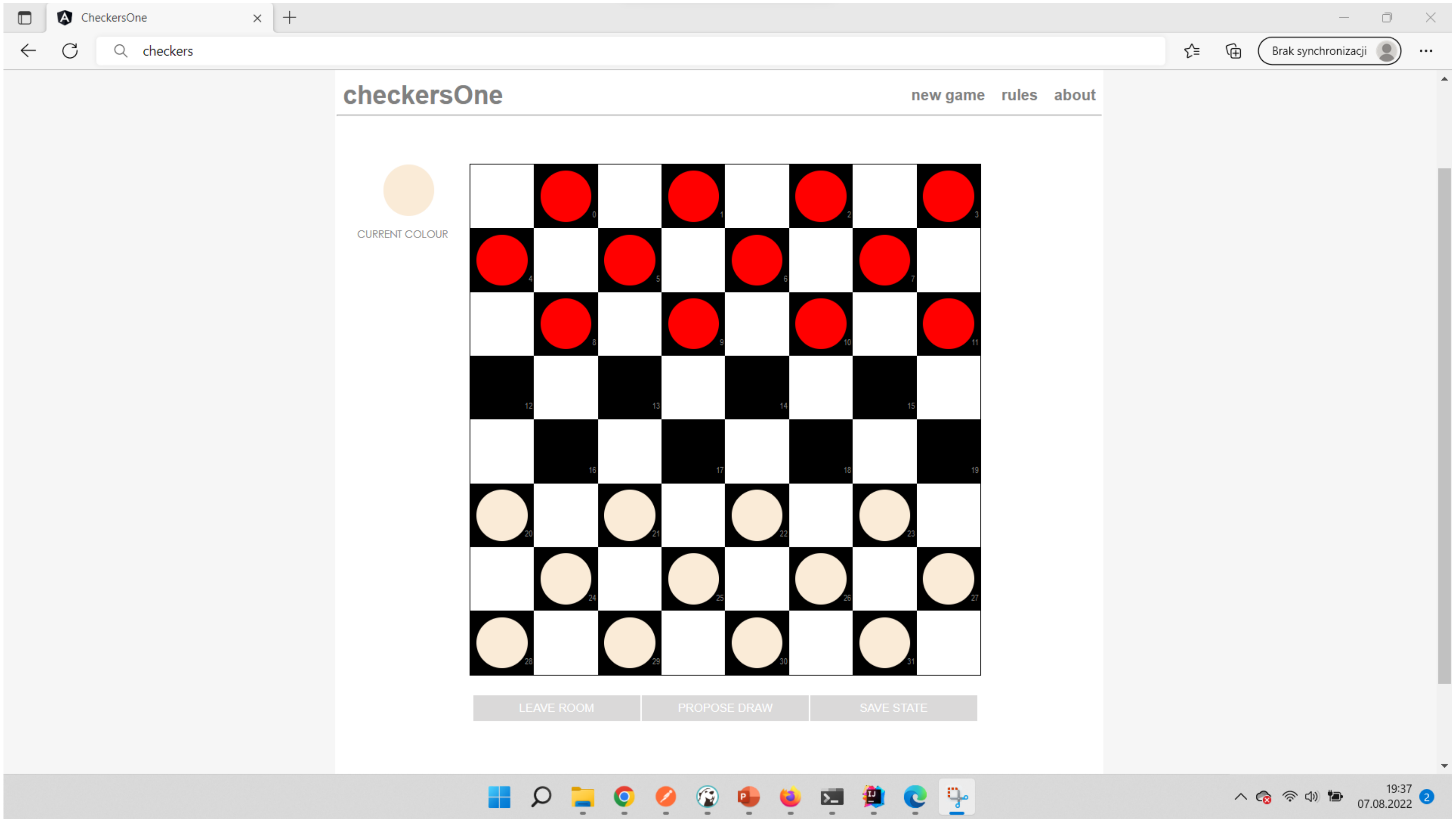

CLIENT
websocket
SERVER
render game state
InitialState: GameState
move: PawnMove
validateMove(move, state)
state is stored on the server side for security reason
updateMultiplayerState(state)
render game state
Either[
errorMsg: MoveValidationError,
state: GameState]

Multiplayer Game

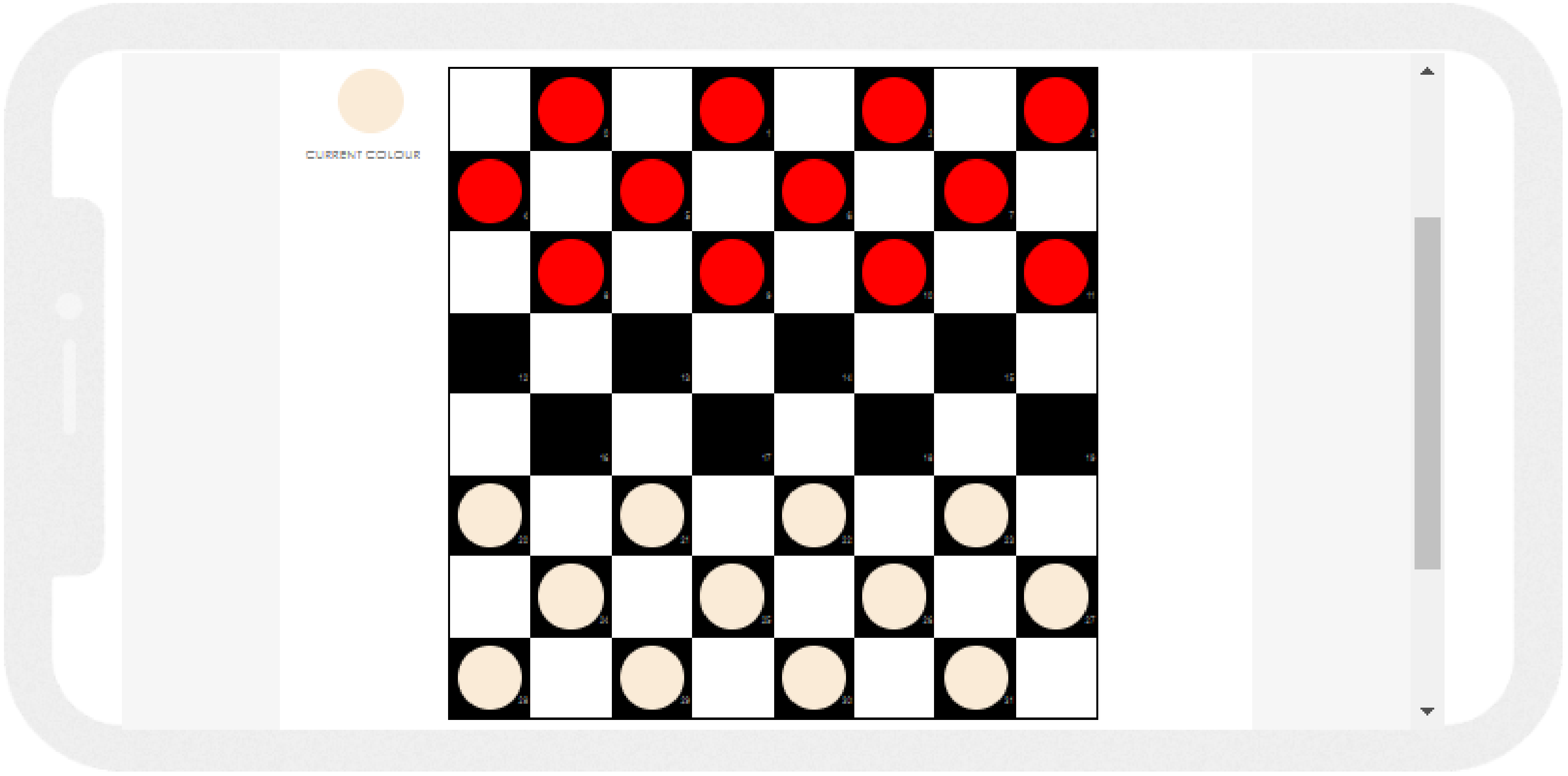


Singleplayer Game

FRONTEND - DESKTOP



FRONTEND - MOBILE



DEMO

TODO

functionalities

- **More advanced AI** (better heuristics, min-max algorithm implementation), configurable difficulty level for single player game
- **User authentication** (players stored in DB) – user stats, personalized game saves, JWT
- **Custom checkers rules** (based on other variants)

technical aspects

- **More tests** for multi player game
- **DB migration scripts** (e.g. using Flyway)
- **PureConfig** for configuration management
- **Cats 3** migration
- **JSON Web Token**

BOOTCAMP - CONCLUSION

+ I did a project that gives me a lot of satisfaction. Thanks to my mentor Marcin for great discussions and code review full of many ideas

+ The Scala Bootcamp helped me better understand functional programming approach

+- There was a lot of informations to be learned during the lectures (sometimes overwhelming). That's good we had an access to the video recording

- Java is not as cool as before (hard to live without for-comprehension or pattern-matching)

THANKS!