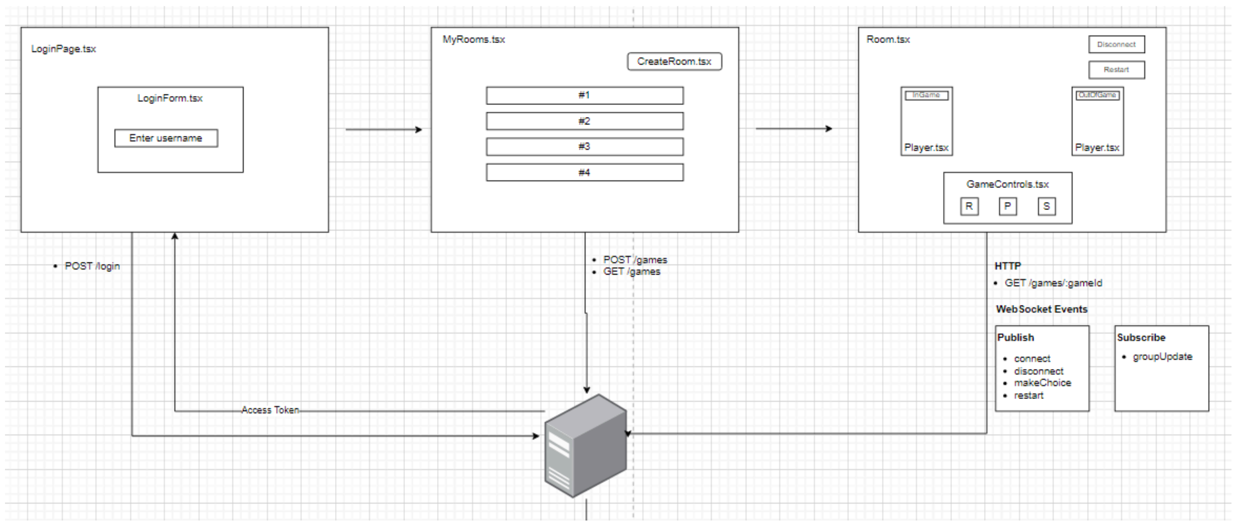
There was a task to develop application that will allow users to solve controversial questions even at a distance - an online version of "Rock, Paper, Scissors"

As main front-end library for easier creation of UI/UX I chose React.

Next steps was to align with requirements that were given to me.

1. The player should be able to enter his username, which will be saved when logging in again
2. A player should be able to see the status of another player (in-game, made a choice, out-of-game)
3. A player should be able to choose one of three elements (rock, paper, scissors) and be able to re-select an element until another player makes their choice
4. A player should be able to start the game from the beginning if his opponent has chosen a piece and the winner of the current game is determined
5. The player should be able to see the result of the game (the score, which is updated after the end of the current game and is reset when one of the players disconnects)

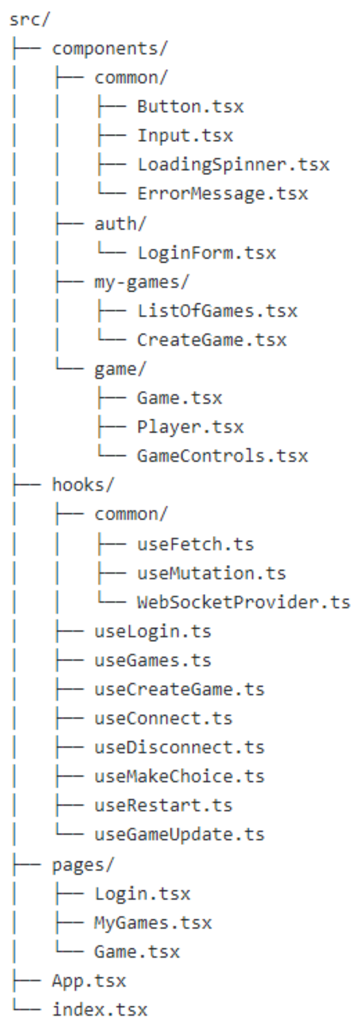
All those requirements were fulfilled within such schematic design:



Since task is not just to develop UI, but to make it functional, we **have to think around architecture and API integration**. Thus, valid project structure, creation of reusable components and proper integration with backend will make our application work accurately and will be flexible in case of new requirements introduction.

Project structure

So, lets start from project structure.



```
src/
├── components/
│   ├── common/
│   │   ├── Button.tsx
│   │   ├── Input.tsx
│   │   ├── LoadingSpinner.tsx
│   │   └── ErrorMessage.tsx
│   ├── auth/
│   │   └── LoginForm.tsx
│   ├── my-games/
│   │   ├── ListOfGames.tsx
│   │   └── CreateGame.tsx
│   └── game/
│       ├── Game.tsx
│       ├── Player.tsx
│       └── GameControls.tsx
├── hooks/
│   ├── common/
│   │   ├── useFetch.ts
│   │   ├── useMutation.ts
│   │   └── WebSocketProvider.ts
│   ├── useLogin.ts
│   ├── useGames.ts
│   ├── useCreateGame.ts
│   ├── useConnect.ts
│   ├── useDisconnect.ts
│   ├── useMakeChoice.ts
│   ├── useRestart.ts
│   └── useGameUpdate.ts
├── pages/
│   ├── Login.tsx
│   ├── MyGames.tsx
│   └── Game.tsx
├── App.tsx
└── index.tsx
```

For **reusable UI** components that may be introduced at other pages, we use src/components/common folder (UI).

For components that should **represent page**, that is composition of more granular components, we use folder src/pages

For components that is **unique to specific page**, we place those inside src/components/* folder. That folder contains UI/Business Logic components that are unique for page and won’t be reused elsewhere (i.e src/components/game Game.tsx,Player.tsx,GameControls)

Since there is also present integration with back-end, there is additional common hooks were introduced. We could have a use of React Query library, but since we don’t need that reach functionality of *optimistic update, caching, etc* that takes a lot of bundle space, we may implement simple alternative that will provide to us feedback regarding request status (to make UI user-friendly) and state updates once data is prepared to be rendered. Example of such is useFetch (for read requests) and useMutation (for CUD requests).

Next is WebSocket integration. Due to real-time collaboration requirement, we have to connect to webserver by WebSocket protocol to achieve two-way communication and react on server published events (i.e gameUpdate). I decided to go with several lines of wrapper around WS API that browser API gives to us. With use of React context, I have created provider that connect to WS server, and with usage of context value I may create custom React hooks that will fulfill requirements (publish and subscribing to events)

Login page

/login page will contain UI with field for username to pass. Then on form submit, we will invoke useLogin (custom hook around useMutation) that will send HTTP POST request to webserver to Login.

If user is already created, server will generate access token and will return it back to user.
If user was not created, server will register user and return access token

On success, redirect to /my-games

My Games page

/my-games page will show list of game that user previously created/was added to.

On landing, FE will make GET /games request to get list of games that possible to connect to

Also, CreateGame.tsx component will provide functionality to create new game (user should choose opponent from list of existing users)

Redirection to /games/:id happens on creation/connection

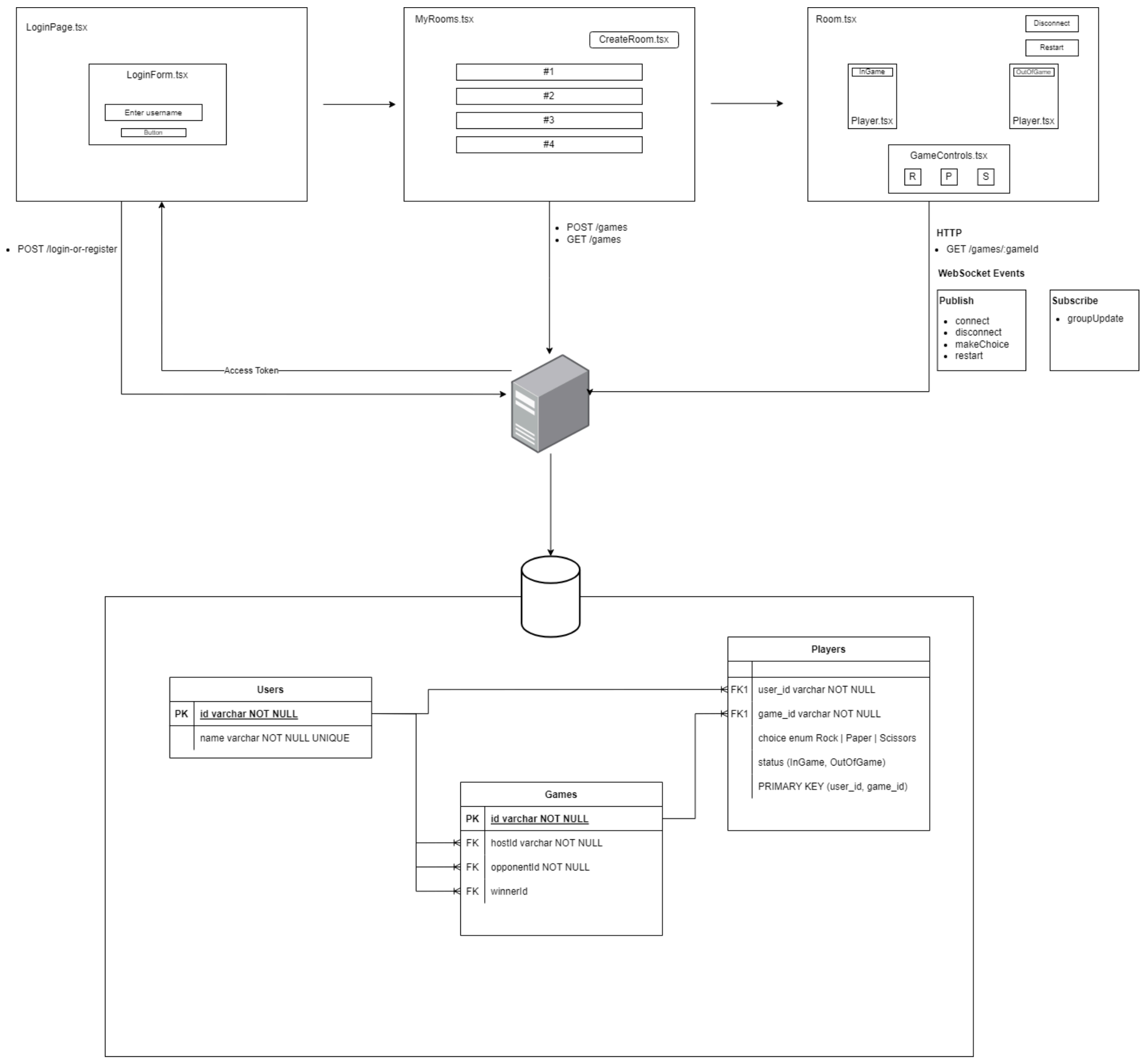
Game page

On landing to /games/:id, WebsocketProvider will initiate websocket connection with server.

Use cases:

1. Connection
    a. Listen from server for event: **gameUpdate** (useGameUpdate)
    b. publish message with event: **connect** (useConnection)
    c. on successful connection gameUpdate will deliver updated content of game
2. Disconnect

    a. publish message with event: **disconnect** (useDisconnect)
    b. Redirect to /my-games page on successful
3. Make Choice
    a. publish message: makeChoice
4. Restart
    a. publish message: restart


LoginPage.tsx
LoginForm.tsx
Enter username
Button
MyRooms.tsx
CreateRoom.tsx
#1
#2
#3
#4
Room.tsx
Disconnect
Restart
InGame
Player.tsx
OutOfGame
Player.tsx
GameControls.tsx
R
P
S
POST /login-or-register
POST /games
GET /games
HTTP
GET /games/:gameId
WebSocket Events
Publish
connect
disconnect
makeChoice
restart
Subscribe
groupUpdate
Access Token
Users
PK id varchar NOT NULL
name varchar NOT NULL UNIQUE
Players
FK1 user_id varchar NOT NULL
FK1 game_id varchar NOT NULL
choice enum Rock | Paper | Scissors
status (InGame, OutOfGame)
PRIMARY KEY (user_id, game_id)
Games
PK id varchar NOT NULL
FK hostId varchar NOT NULL
FK opponentId NOT NULL
FK winnerId