Marco Monteiro & Ricardo Gomes

Worksheet 6 – Multiplayer

Desenvolvimento de Aplicações Distribuídas

Practical Worksheet Departamento Engenharia Informática

Objective

This worksheet will allow us to start using the Web Socket Server (Socket.io) to implement a multiplayer game.

Prerequisite

This section builds upon everything covered in **worksheets 4 and 5**, where we implemented the REST API, including data persistence, authentication, data transformation, file upload, and authorization.

If you are starting from the provided solution for worksheet 5, instead of continuing with your own version of the project, begin by installing the required PHP packages, then recreate the database and the public storage link by executing the following commands on the "api" folder:

```
composer update
php artisan migrate:fresh --seed
php artisan storage:link
```

1 Join Web Socket Server

Our application will only allow authenticated users to use the Multiplayer feature, so the first thing to set up is to tie the user login to the web socket server.

Before proceeding, we'll enhance the WebSocket server setup by adding ESLint for code quality enforcement and Nodemon[1] for automatic server restarts during development.

1. Install the following packages on the `websockets` folder

```
npm install --save-dev eslint @eslint/js globals nodemon
```

2. Run the following command to set up the base config for ESLint

```
npx eslint --init
```

3. Change the `globals.browser` parameter to `globals.node` on the `websockets/esling.config.js` file
4. Change the dev script on the `websockets/package.js` to `dev: nodemon index.js`
5. Create two folders – `events` and `state` - on the `websockets` folder
6. Add the provided `server.js` file to the `websockets` folder:

[1] https://nodemon.io/

7. Change the contents of `websocket/index.js` to the following:

```
import { serverStart } from "./server.js"

const PORT = process.env.PORT || 3000

serverStart(PORT)

console.log(`Socket.io server running on port ${PORT}`)
console.log("Waiting for connections...")
```

8. Create two files, both called `connection.js` in the `websockets/events` and `websockets/state` folders.

To keep track of users we will set up a Map that users the user ID as a key and holds the ID of the socket and the user object.

9. Add the following code to the `websockets/state/connection.js` file:

```
const users = new Map()

export const addUser = (socket, user) => {
      users.set(socket.id, user)
}

export const removeUser = (socketID) => {
      const userToDelete = { ... users.get(socketID)}
      users.delete(socketID)
      return userToDelete
}

export const getUser = (socketID) => {
  return users.get(socketID)
}

export const getUserByID = (userID) => {
  for (let [socketID, user] of users.entries()) {
    if (user.id == userID) {
      return {
        user,
        socketID,
      }
    }
  }
  return null
}

export const getUserCount = () => {
  return users.size
}
```

10. Add the following code to the `websockets/events/connection.js` file:

```
import { addUser, removeUser, getUserCount } from "../state/connection.js"

export const handleConnectionEvents = (io, socket) => {
	socket.on("join", (user) => {
		addUser(socket, user)
		console.log(`[Connection] User ${user.name} has joined the server`)
		console.log(`[Connection] ${getUserCount()} users online`)
		io.emit("player-joined", {
			socketID: socket.id,
			userID: user.id,
			userName: user.name,
		})
	})

	socket.on("leave", () => {
		const user = removeUser(socket.id)
		console.log(`[Connection] User ${user.name} has left the server`)
		console.log(`[Connection] ${getUserCount()} users online`)
	})

	socket.on("disconnect", () => {
		console.log("Connection Lost:", socket.id)
		const user = removeUser(socket.id)
		console.log(`[Connection] ${getUserCount()} users online`)
	})
}
```

Next we need to change the way our VueJS application handles the sockets, let's start by creating a Pinia store for the sockets

11. Create a new file in `frontend/src/stores/socket.js` and add the following contents:

```
import { defineStore } from 'pinia'
import { inject, ref } from 'vue'
import { useAuthStore } from './auth'

export const useSocketStore = defineStore('socket', () => {
    const socket = inject('socket')
    const authStore = useAuthStore()

    const joined = ref(false)

    const emitJoin = (user) => {
        if (joined.value) return
        console.log(`[Socket] Joining Server`)
        socket.emit('join', user)
        joined.value = true
    }

    const emitLeave = () => {
        socket.emit('leave')
        console.log(`[Socket] Leaving Server`)
        joined.value = false
    }

    const handleConnection = () => {
      socket.on('connect', () => {
          console.log(`[Socket] Connected -- ${socket.id}`)
          if (authStore.isLoggedIn && !joined.value) {
                emitJoin(authStore.currentUser)
            }
        })

        socket.on('disconnect', () => {
            joined.value = false
            console.log(`[Socket] Disconnected -- ${socket.id}`)
        })
    }

    return {
      emitJoin,
      emitLeave,
      handleConnection,
    }
})
```

12. Replace our `frontend/src/App.vue` file with the one provided
13. On the Auth store call the `emitJoin` and `emitLeave` on the `login` and `logout` methods.
14. Test the application:
    14.2. Make sure we are using nodemon on the web sockets by stopping and running `npm run dev`
    14.3. Open two instances of our application in a browser (or different browsers)
    14.4. Try the login (e.g., of emails: test@example.com; dad_user1@example.com; dad_user2@example.com) and log out feature and review the output of the web socket server

2 Game Lobby

Our next tasks are to allow users to create multiplayer games, and have other users see those games.

15. Add the provided `MultiplayerLobbyPage.vue` and `MultiplayerGamePage.vue` files to the `frontend/src/pages/games` folder
16. On the `frontend/src/router/index.js` implement the following code (if it is not yet implemented) and review its contents, in particular the `router.beforeEach` call

```
import MultiplayerLobbyPage from '@/pages/game/MultiplayerLobbyPage.vue'
import MultiplayerGamePage from '@/pages/game/MultiplayerGamePage.vue'
import { useAuthStore } from '@/stores/auth'
import { toast } from 'vue-sonner'

// New Routes-  CHILDREN OF path: '/games',
{
    path: 'lobby',
    name: 'multiplayer-lobby',
    component: MultiplayerLobbyPage,
    meta: { requiresAuth: true },
},
{
    path: 'multiplayer',
    name: 'multiplayer',
    component: MultiplayerGamePage,
    meta: { requiresAuth: true },
},


// Route Guards
router.beforeEach((to, from, next) => {
    const authStore = useAuthStore()
    if (to.meta.requiresAuth && !authStore.isLoggedIn) {
        toast.error('This navigation requires authentication')
        next({ name: 'login' })
    } else {
        next()
    }
})
```

17. On the `frontend/src/stores/socket.js` add (and return) the following methods:

```
import { useGameStore } from './game'
// ....
const gameStore = useGameStore()
// ....

const emitGetGames = () => {
    socket.emit('get-games')
}

const handleGameEvents = () => {
    socket.on('games', (games) => {
        console.log(`[Socket] server emited games | game count ${games.length}`)
        gameStore.setGames(games) // Import and instantiate the game store
    })
}
```

18. On the `frontend/src/App.vue` file add the call to the new `handleGameEvents` method on the onMounted function – uncomment the corresponding line.
19. Add the following code to the `frontend/src/stores/game.js` file, and return the new elements:

```
import { toast } from 'vue-sonner'
// ...
const socket = inject('socket')

const games = ref([])

const createGame = (difficulty = 'medium') => {
    if (!authStore.currentUser) {
        toast.error('You must be logged in to create a game')
        return
    }
    if (!socket || !socket.connected) {
        toast.error('Not connected to server. Please refresh the page.')
        return
    }
    socket.emit('create-game', difficulty)
}

const setGames = (newGames) => {
    games.value = newGames
    console.log(`[Game] Games changed | game count ${games.value.length}`)
}

const myGames = computed(() => {
    return games.value.filter((game) => game.creator == authStore.currentUser.id)
})

const availableGames = computed(() => {
    return games.value.filter((game) => game.creator != authStore.currentUser.id)
})
```

20. On the `frontend/src/pages/game/MultiplayerLobbyPage.vue`:

20.2. Change the local reactive variables (myGames and availableGames) on the template, with the ones provided by the game store. Remove these two local reactive variables.

20.3. Call the socket store `emitGetGames` from the `onMounted` callback

20.4. Implement the `createNewGame` method by calling the game store `createGame` function and passing the currently selected difficulty

21. Add two new files to the WebSockets project, both called `game.js` to the `websockets/events` and `websockets/state` folders.
22. Add this code to the `websockets/events/game.js` file and review it:

```
import { getUser } from "../state/connection.js"
import { createGame, getGames } from "../state/game.js"

export const handleGameEvents = (io, socket) => {
    socket.on("create-game", (difficulty) => {
        const user = getUser(socket.id)
        const game = createGame(difficulty, user)
        socket.join(`game-${game.id}`)
        console.log(`[Game] ${user.name} created a new game - ID: ${game.id}`)
        io.emit("games", getGames())
    })

    socket.on("get-games", () => {
        io.emit("games", getGames())
    })
}
```

23. Add this code to the `websockets/state/game.js` file and review it:

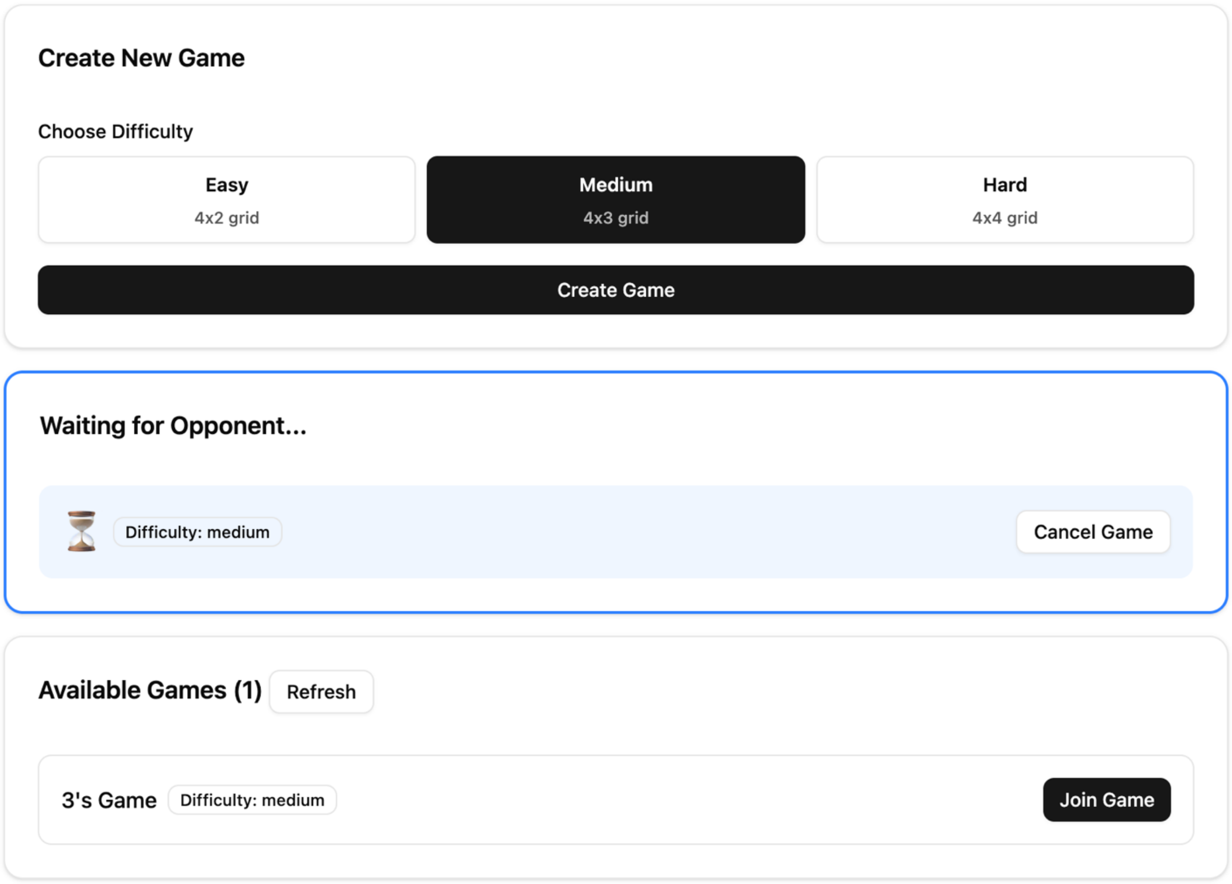
```
const games = new Map()
let currentGameID = 0

export const createGame = (difficulty, user) => {
    currentGameID++
    const game = {
        id: currentGameID,
        difficulty,
        creator: user.id,
        player1: user.id,
        player2: null,
    }
    games.set(currentGameID, game)
    return game
}

export const getGames = () => {
    return games.values().toArray()
}
```

24. Call the `handleGameEvents(io, socket)` method on the `websockets/index.js` `on('connection')` handler – uncomment the references to handleGameEvents.

25. Test the application with two browsers, you should see something like this after both users created games:



3 Multiplayer

Now let's implement support for the multiplayer games.

26. Add the provided `MultiplayerGamePage.vue` to the `frontend/src/pages/game/` folder
27. Add, and return the following method to the `frontend/src/stores/auth.js` file:

```
const currentUserID = computed(() => {
    return currentUser.value?.id
})
```

28. Add, and return, the following elements to the `frontend/src/stores/socket.js` file:

```
const emitJoinGame = (game) => {
    console.log(`[Socket] Joining Game ${game.id}`)
    socket.emit('join-game', game.id, authStore.currentUser.id)
}

const emitFlipCard = (gameID, card) => {
    socket.emit('flip-card', gameID, card)
}
```

29. Add the following socket event listener to the `handleGameEvents` method of `frontend/src/stores/socket.js`:

```
socket.on('game-change', (game) => {
    gameStore.setMultiplayerGame(game)
})
```

30. Add (or replace) the following methods to the `frontend/src/pages/game/MultiplayerLobbyPage.vue`:

```
import { useRouter } from 'vue-router'

const router = useRouter()
// ...

const joinGame = (game) => {
    socketStore.emitJoinGame(game)
}

const startGame = (game) => {
    gameStore.multiplayerGame = game
    router.push({ name: 'multiplayer' })
}
```

31. Add, and return the following elements to the `frontend/src/stores/game.js` file:

```
const multiplayerGame = ref({})

const setMultiplayerGame = (game) => {
    multiplayerGame.value = game
    console.log(`[Game] Multiplayer Game changed | game moves ${game.moves}`)
}
```

Our VueJS application should now be set to handle the multiplayer game, but we still need to implement support for them in the Web Sockets Server

Let's start by the events

32. Implement these socket event handlers on the `websockets/events/game.js` file:

```
import { createGame, getGames, joinGame, flipCard, clearFlippedCard} from
"../state/game.js"

socket.on("join-game", (gameID, userID) => {
	joinGame(gameID, userID)
	socket.join(`game-${gameID}`)
	console.log(`[Game] User ${userID} joined game ${gameID}`)
	io.emit("games", getGames())
	io.to(`game-${gameID}`).emit("game-ready")
})

socket.on("flip-card", (gameID, card) => {
	const game = flipCard(gameID, card)
	io.to(`game-${gameID}`).emit("game-change", game)
})
```

33. Answer the following questions:
    - 33.2. What does the `socket.join`(game-${gameID})`;` instruction do?
    - 33.3. We have two calls to `io.to`. What is their purpose?

Now we need to implement the game logic. Let's start by the creation of the game itself

34. Add these elements to the `websockets/state/game.js` file:

```
let currentGameID = 0
const options = [1, 2, 3, 4, 5, 6, 7, 8].map((i) => {
	return { face: i, matched: false, flipped: false }
})

const generateBoard = (difficulty) => {
	const cards = []
	let numPairs = 4

	if (difficulty === "medium") numPairs = 6
	if (difficulty === "hard") numPairs = 8

	const boardOptions = options.slice(0, numPairs)
	let idCounter = 0

	boardOptions.forEach((option) => {
		cards.push({ id: idCounter++, ...option })
		cards.push({ id: idCounter++, ...option })
	})

	for (let i = cards.length - 1; i > 0; i--) {
		const j = Math.floor(Math.random() * (i + 1));
		[cards[i], cards[j]] = [cards[j], cards[i]]
	}

	return cards
}
```

35. Now let’s change the `createGame` method to use these new features:

```
export const createGame = (difficulty, user) => {

       currentGameID++
       const game = {
              id: currentGameID,
              difficulty,
              creator: user.id,
              player1: user.id,
              player2: null,
              winner: null,
              currentPlayer: user.id,
              cards: generateBoard(difficulty),
              flippedCards: [],
              matchedPairs: [],
              started: false,
              complete: false,
              moves: 0,
              beganAt: null,
              endedAt: null,
       }

       games.set(currentGameID, game)
       return game
}
```

Now the support for player 2 joining the game

36. Add this method to the game state:

```
export const joinGame = (gameID, player2) => {
    games.get(gameID).player2 = player2
}
```

Now the methods to handle the card flips.

37. Add these methods to the game state:

```
export const flipCard = (gameID, card) => {
	const game = games.get(gameID)

	if (!game.beganAt) {
		game.beganAt = new Date()
		game.started = true
	}

	const gameCard = game.cards.find((c) => c.id == card.id)
	if (game.flippedCards.includes(gameCard.id)) return
	if (game.matchedPairs.includes(gameCard.id)) return
	if (game.flippedCards.length >= 2) return

	game.flippedCards.push(gameCard.id)
	gameCard.flipped = true
	if (game.flippedCards.length == 2) {
		game.moves++
		checkForMatch(game)
		checkForGameComplete(game)
	}

	return game
}

const checkForMatch = (game) => {

	if (game.flippedCards.length !== 2) return

	const [first, second] = game.flippedCards
	const firstCard = game.cards.find((c) => c.id === first)
	const secondCard = game.cards.find((c) => c.id === second)

	if (firstCard.face === secondCard.face) {
		game.matchedPairs.push(first, second)
		firstCard.matched = true
		secondCard.matched = true
		game.flippedCards = []
	} else {
		firstCard.flipped = true
		secondCard.flipped = true
		setTimeout(() => {
			triggerFlipDelay(game)
		}, 1000)
	}
}

const checkForGameComplete = (game) => {

	if (game.matchedPairs.length === game.cards.length) {
		game.complete = true
		game.winner = game.currentPlayer
		game.endedAt = new Date()
	}
}
```

```
export const clearFlippedCard = (game) => {

       if (game.flippedCards.length !== 2) return

       const [first, second] = game.flippedCards
       const firstCard = game.cards.find((c) => c.id === first)
       const secondCard = game.cards.find((c) => c.id === second)

       firstCard.flipped = false
       secondCard.flipped = false
       game.flippedCards = []
       game.currentPlayer = game.currentPlayer == game.player1 ? game.player2 :
game.player1

       return game
}
```

Did you notice the setTimeout on the `checkForMatch` method? This is how we are handling the delay for the card flipping, in essence:

- The user clicks the card and sends an emit to the server
- The server handles the flip and emits a changed version of the game (both users see the flip)
- If the flipped card is the second, the server first emits the game with both cards flipped
- If the cards did not match the server waits 1 second and then sends the board back with cards hidden again

38. Add this code to the `websockets/events/game.js` file:

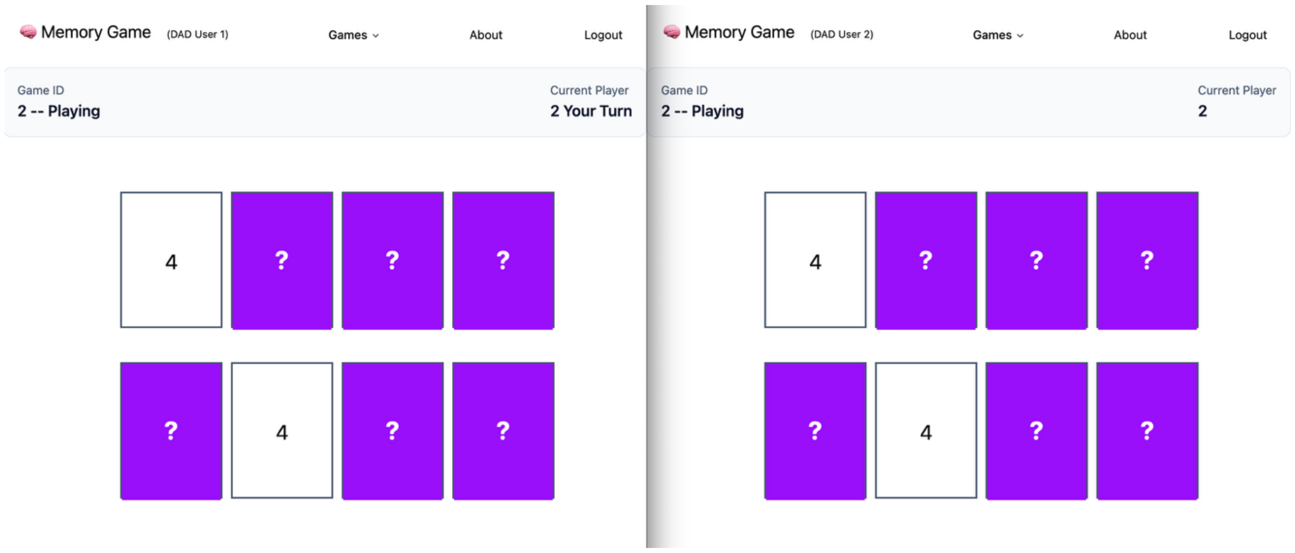
```
import { server } from "../server.js";

export const triggerFlipDelay = (game) => {
    game = clearFlippedCard(game);
    server.io.to(`game-${game.id}`).emit("game-change", game);
};
```

39. Add this code (to import the triggerFlippedDelay) to the `websockets/state/game.js` file:

```
import { triggerFlipDelay } from "../events/game.js"
```

40. The game should be playable now, and it should look something like this:



4 Final Adjustments (optional)

Try the multiplayer game and find and solve any bug detected. Also, make some final adjustments for the game to support "cancel game", to replace the start button with a "continue" button when the game is running and to remove it when no longer needed, to remove the game from the lobby when it is finished, etc.