React State With Fetch

A more complex example

Tic Tac Toe With an Expert API

Step 1 - Static Implementation

```
import React from 'react'

export function App() {
  return (
    <div>
      <h1>
        Tic Tac Toe - <button>New</button>
      </h1>
      <ul>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
      </ul>
    </div>
  )
}
```

Step 1 - ALL THE CSS

```css
:root {
  /* CSS Variables for all the font colors and sizes. Try changing these! */
  --header-background: #5661b3;
  --header-text-color: #fff9c2;
  --header-font-size: 2rem;
  --square-font-size: calc(8 * var(--header-font-size));
  --square-text-color: #5661b3;
  --square-background-color: #e0e8ff;
  --square-border: 3px solid var(--square-text-color);

  font: 16px / 1 sans-serif;
}

html {
  height: 100%;
}

body {
  margin: 0;
  min-height: 100%;
}

h1 {
  /* center the header */
  text-align: center;

  /* Use a sans serif font with a little spacing and color */
  font-family: Verdana, Geneva, Tahoma, sans-serif;
  letter-spacing: 0.4rem;
  font-size: var(--header-font-size);
  color: var(--header-text-color);

  /* Remove margins and set a little padding */
  margin: 0;
  padding: var(--header-font-size);

  /* Set a background color for the header */
  background-color: var(--header-background);
}

ul,
li {
  /* Be gone margins! */
  margin: 0;
  padding: 0;

  /* and list styles */
  list-style: none;
}

ul {
  /* Make the height of the list equal to the height of the page MINUS the height taken by the header */
  height: calc(100vh - 3 * var(--header-font-size));

  /* Display the list as a 3 column and three row grid */
  display: grid;
  grid-template: 1fr 1fr 1fr / 1fr 1fr 1fr;

  /* Add a little gap between to allow the background color through */
  gap: 1rem;

  /* Set the background color that will show through the gap */
  background-color: var(--square-text-color);
}

ul li {
  /* Use a monospace font */
  font-family: monospace;
  font-size: var(--square-font-size);

  /* Style the background color of the item */
  background-color: var(--square-background-color);

  /* Make the cursor a pointer by default */
  cursor: pointer;

  /* Center the text in the LI */
  display: flex;
  align-items: center;
  justify-content: center;

  /* Don't let the squares become too small */
  min-width: 3rem;
  min-height: 10rem;
}

ul li.taken {
  cursor: not-allowed;
}

ul li.small {
  font-size: 4rem;
}

ul li.not-allowed-click {
  background-color: red;
}
```

Step 2: Make a state using data

- When using an API, taking the data example from the API is a great way to start.


```
{
  "board": [
    [
      " ",
      " ",
      " "
    ],
    [
      " ",
      " ",
      " "
    ],
    [
      " ",
      " ",
      " "
    ]
  ],
  "winner": null,
}
```

Step 2 Continued:

```
import React, { useState } from 'react'

export function App() {
  const [game, setGame] = useState({
    board: [
      [' ', ' ', ' '],
      [' ', ' ', ' '],
      [' ', ' ', ' '],
    ],
    id: null,
    winner: null,
  })

  return (
    <div>
      <h1>
        Tic Tac Toe - <button>New</button>
      </h1>
      <ul>
        <li>{game.board[0][0]}</li>
        <li>{game.board[0][1]}</li>
        <li>{game.board[0][2]}</li>
        <li>{game.board[1][0]}</li>
        <li>{game.board[1][1]}</li>
        <li>{game.board[1][2]}</li>
        <li>{game.board[2][0]}</li>
        <li>{game.board[2][1]}</li>
        <li>{game.board[2][2]}</li>
      </ul>
    </div>
  )
}
```

Step 3: Try manually changing the state

```
const [game, setGame] = useState({
  board: [
    [' ', ' ', 'O'],
    [' ', ' ', ' '],
    ['X', ' ', ' '],
  ],
  id: null,
  winner: null,
})
```

See that this affects the user interface

Step 4: Connect the actions

- Define a method that will handle clicking on the cell
- We will need to know the row and column

```
function handleClickCell(row: number, column: number) {
  console.log(`You clicked on row ${row} and column ${column}`)
}
```

```tsx
import React, { useState } from 'react'

export function App() {
  const [game, setGame] = useState({
    board: [
      [' ', ' ', ' '],
      [' ', ' ', ' '],
      [' ', ' ', ' '],
    ],
    id: null,
    winner: null,
  })

  function handleClickCell(row: number, column: number) {
    console.log(`You clicked on row ${row} and column ${column}`)
  }

  return (
    <div>
      <h1>
        Tic Tac Toe - <button>New</button>
      </h1>
      <ul>
        <li onClick={() => handleClickCell(0, 0)}>{game.board[0][0]}</li>
        <li onClick={() => handleClickCell(0, 1)}>{game.board[0][1]}</li>
        <li onClick={() => handleClickCell(0, 2)}>{game.board[0][2]}</li>
        <li onClick={() => handleClickCell(1, 0)}>{game.board[1][0]}</li>
        <li onClick={() => handleClickCell(1, 1)}>{game.board[1][1]}</li>
        <li onClick={() => handleClickCell(1, 2)}>{game.board[1][2]}</li>
        <li onClick={() => handleClickCell(2, 0)}>{game.board[2][0]}</li>
        <li onClick={() => handleClickCell(2, 1)}>{game.board[2][1]}</li>
        <li onClick={() => handleClickCell(2, 2)}>{game.board[2][2]}</li>
      </ul>
    </div>
  )
}
```

Step 5: Update the state

- For this, we will use the Tic Tac Toe API
- The first step is to make a new game by clicking on new

```
async function handleNewGame() {
  // Make a POST request to ask for a new game
  const response = await fetch(
    'https://sdg-tic-tac-toe-api.herokuapp.com/game',
    {
      method: 'POST',
      headers: { 'content-type': 'application/json' },
    }
  )

  if (response.ok) {
    // Get the response as JSON
    const newGameState = await response.json()

    // Make that the new state!
    setGame(newGameState)
  }
}
```

```
<h1>
  Tic Tac Toe - <button onClick={handleNewGame}>New</button>
</h1>
```

See that this updates the user interface

- Test this by making the elements in the initial `board` state contain something other than spaces!
- Creating a new game should visually "reset" the board

Notice the state has some extra information in it now

- We know the *id* of the game. Useful for future API requests.


```
{
  "id": 5,
  "board": [
    [
      " ",
      " ",
      " "
    ],
    [
      " ",
      " ",
      " "
    ],
    [
      " ",
      " ",
      " "
    ]
  ],
  "winner": null,
  "created_at": "2021-02-19T00:52:49.678Z",
  "updated_at": "2021-02-19T00:52:49.678Z"
}
```

Update handleClickCell

```
async function handleClickCell(row: number, column: number) {
  // Generate the URL we need
  const url = `https://sdg-tic-tac-toe-api.herokuapp.com/game/${game.id}`

  // Make an object to send as JSON
  const body = { row: row, column: column }

  // Make a POST request to make a move
  const response = await fetch(url, {
    method: 'POST',
    headers: { 'content-type': 'application/json' },
    body: JSON.stringify(body),
  })

  if (response.ok) {
    // Get the response as JSON
    const newGameState = await response.json()

    // Make that the new state!
    setGame(newGameState)
  }
}
```

Handle the winner

- The API gives us information about the winner.
- Let us make the header display the winner

Dynamically generate the header

```
const header = 'Tic Tac Toe'

return (
  <div>
    <h1>
      {header} - <button onClick={handleNewGame}>New</button>
    </h1>
```

Now make it depend on the winner state

```
const header = game.winner ? `${game.winner} is the winner` : 'Tic Tac Toe'
```

Remove duplication in the creation of the game board

- Let us use `map` instead of repeating all the `li`

```
<ul>
  {game.board.map((boardRow, rowIndex) => {
    return boardRow.map((cell, columnIndex) => {
      return (
        <li
          key={columnIndex}
          onClick={() => handleClickCell(rowIndex, columnIndex)}
        >
          {cell}
        </li>
      )
    })
  })}
</ul>
```

Block clicks

- When there is no game
- Or when the user clicks on an occupied cell
- Or when someone has won

```
if (
  // No game id
  game.id === undefined ||
  // A winner exists
  game.winner ||
  // The space isn't blank
  game.board[row][column] !== ' '
) {
  return
}
```

Dynamically set the class name

If the cell is not empty, set the class to taken

```
<ul>
  {game.board.map((boardRow, rowIndex) => {
    return boardRow.map((cell, columnIndex) => {
      return (
        <li
          key={columnIndex}
          className={cell === ' ' ? undefined : 'taken'}
          onClick={() => handleClickCell(rowIndex, columnIndex)}>
          {cell}
        </li>
      )
    })
  })}
</ul>
```

Steps:

- Step 1 - Static implementation
- Step 2 - Make a state object containing data
- Step 3 - Try manually changing the value in the state.
- Step 4 - Connect actions
- Step 5 - Update state
- Step 5a - Use fetch to send required data to the API
- Step 5b - Use the response from fetch to get the new state

Refining our TypeScript

- `response.json()` returns any!
- The data could be: `number | string| boolean | null | object | Array`
- TypeScript has no way to know.
- We can define some types to get *some* type checking

Game state

```
type Game = {
  board: [
    ['X' | 'O' | ' ', 'X' | 'O' | ' ', 'X' | 'O' | ' '],
    ['X' | 'O' | ' ', 'X' | 'O' | ' ', 'X' | 'O' | ' '],
    ['X' | 'O' | ' ', 'X' | 'O' | ' ', 'X' | 'O' | ' ']
  ]
  id: null | number
  winner: null | string
}
```

Game state with type for each square

```
type Square = 'X' | 'O' | ' '

type Game = {
  board: [
    [Square, Square, Square],
    [Square, Square, Square],
    [Square, Square, Square]
  ]
  id: null | number
  winner: null | string
}
```

Row, Row, Row (your ⛵)

```
type Square = 'X' | 'O' | ' '
type Row = [Square, Square, Square]

type Game = {
  board: [Row, Row, Row]
  id: null | number
  winner: null | string
}
```

Board type for game state

```
type Square = 'X' | 'O' | ' '
type Row = [Square, Square, Square]
type Board = [Row, Row, Row]

type Game = {
  board: Board
  id: null | number
  winner: null | string
}
```

Using game state

```
const [game, setGame] = useState<Game>({
  board: [
    [' ', ' ', ' '],
    [' ', ' ', ' '],
    [' ', ' ', ' '],
  ],
  id: null,
  winner: null,
})
```

Using game state

```
const newGameState = (await response.json()) as Game
```

Warning!

- `TypeScript` only checks types while in development mode!
- If the API returns something *NOT* in the shape of a `Game` we will have no safety.
- We'd have to write our own validation functions (and we might want to do that if we want to be 💯 sure)
- This might be improved in future versions of TypeScript.