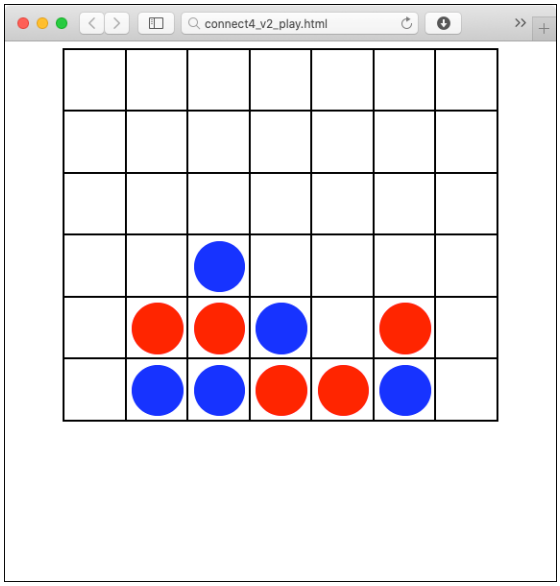
WBE-Praktikum 12

UI-Bibliothek (Teil 2)

Aufgabe 1: «Vier gewinnt» mit Komponenten (Miniprojekt)

Das Spiel «Vier gewinnt» soll nun mit Hilfe von Komponenten und einer auf React basierenden UI-Bibliothek umgesetzt werden.



Folgende Komponenten sind vorgesehen:

- **App**: Container-Komponente, welche für die Anzeige die *Board*-Komponente verwendet.
- **Board**: diese Komponente gibt das Spielfeld aus und verwendet dazu die *Field*-Komponente.
- **Field**: dient zur Ausgabe eines Felds (Quadrat) mit blauer oder roter Spielfigur (oder leer).

Am besten gehen Sie von der letzten Version des Programms aus, in der das Spielfeld noch mit Hilfe der Funktion *renderSJDON* ausgegeben wurde.

Hinweise zur UI-Bibliothek

Entsprechend der Demos und Slides im Unterricht gibt es hier verschiedene Optionen. Es ist möglich, direkt React zu verwenden, als Notation ist auch mit React sowohl JSX als auch die Eigenentwicklung SJDON möglich. Ebenfalls möglich ist es, eine Version der Eigenentwicklung (SuiWeb) zu verwenden. Beispiele für die verschiedenen Möglichkeiten finden Sie in den Demos.

Für die Entwicklung könnte es von Vorteil sein, die App zunächst einmal so zu entwickeln, dass sie «standalone» läuft, d.h. kein Build-Step nötig ist. Das ist problemlos möglich. Selbst JSX lässt sich mit einer Standalone-Version von Babel direkt verarbeiten. Wichtig: dieses Vorgehen ist ungeeignet für die endgültige Version einer Software. Aber für das kleine Projekt in diesem Praktikum geht es vor allem um Möglichkeiten zum Experimentieren mit dem komponentenbasierten Ansatz und nicht um die Performanz der Software.

Die beiden Komponenten *App* und *Board* können folgendermassen implementiert werden:

```
const App = () => [Board, {board: state.board}]

const Board = ({board}) => {
  let flatBoard = [].concat(...board)
  let fields = flatBoard.map((type) => [Field, {type}])
  return (
    ["div", {className: "board"}, ...fields]
  )
}
```

- Studieren Sie den Aufbau der *App*- und der *Board*-Komponente. Wie wird das Array der Fields angelegt? Was bewirkt das Code-Stück `[].concat(...board)`? Probieren Sie es bei Bedarf in der interaktiven Node.js-Shell aus.
- Implementieren Sie die *Field*-Komponente:

  ```
  const Field = ({type}) => {
    ...
  }
  ```

  Sie bekommt einen Typ ('r', 'b' oder '') und soll eine Struktur wie diese erzeugen:

  ```
  <div class="field">
    <div class="piece red"></div>
  </div>
  ```

  Analog mit *blue* für eine blaue Spielfigur. Wenn das Feld leer ist, fehlt das innere *div*.
  Hinweis: statt *class* muss als Attributbezeichnung *className* verwendet werden.

Aufgabe 2: Unveränderliche Datenstrukturen

Das in den letzten Praktikumslektionen und in Aufgabe 1 implementierte Spiel verwaltet den aktuellen Spielstand in einer globalen Variablen *state*, welche alle Informationen über den aktuellen Zustand enthält. Dieses Zustandsobjekt wird aktualisiert, indem einzelne Attributwerte oder Stellen im *board*-Array überschrieben werden.

Das Zustandsobjekt wird also an Ort und Stelle angepasst. Das ist nicht in jedem Falle erwünscht. Daher soll in dieser Aufgabe die Möglichkeit geschaffen werden, Objekte und Arrays *nicht-destruktiv* anzupassen. In der nächsten Aufgabe wird dann gezeigt, wie diese Vorgehensweise genutzt werden kann, um das «Vier gewinnt»-Spiel um eine *Undo*-Funktion zu erweitern.

- Implementieren Sie eine Funktion *setInList(lst, idx, val)*, welche in einem Array *lst* das Element an der Position *idx* mit dem Wert *val* belegt, ohne das ursprüngliche Objekt zu verändern. Es wird also ein neues Array zurückgegeben. Es soll aber keine tiefe Kopie des Arrays erstellt werden: bestehende Referenzen bleiben erhalten. Zum Beispiel:

```
> let lista = [0, 1, [2, 3], 4, {a: 1}]
undefined
> let listb = setInList(lista, 3, 99)
undefined
> lista
[ 0, 1, [ 2, 3 ], 4, { a: 1 } ]
> listb
[ 0, 1, [ 2, 3 ], 99, { a: 1 } ]
> lista[2]
[ 2, 3 ]
> lista[2] === listb[2]
true
> lista[4] === listb[4]
true
```

- Implementieren Sie eine Funktion *setInObj(obj, attr, val)*, welche in einem Objekt *obj* das Attribut *attr* mit dem Wert *val* belegt, ohne das ursprüngliche Objekt zu verändern. Es wird also ein neues Objekt zurückgegeben. Es soll aber keine tiefe Kopie des Objekts erstellt werden. Beispiel:

```
> let obja = { a: {a:1}, b: 5, c: [1,2,3] }
undefined
> let objb = setInObj(obja, "b", 99)
undefined
> obja
{ a: { a: 1 }, b: 5, c: [ 1, 2, 3 ] }
> objb
{ a: { a: 1 }, b: 99, c: [ 1, 2, 3 ] }
> obja.a === objb.a
true
> obja.c === objb.c
true
```

Damit haben wir eine einfache Möglichkeit geschaffen, bestehende Datenstrukturen unverändert zu lassen, auch wenn Mutationen erforderlich sind. Unveränderbare Datenstrukturen sind häufig hilfreich. Die Facebook-Entwickler stellen mit **Immutable.js**[1] eine JavaScript-Bibliothek zur Verfügung, um mit solchen unveränderbaren Datenstrukturen zu arbeiten. Die moderne Programmiersprache Clojure[2] setzt ebenfalls auf unveränderbare Datenstrukturen.

[1] https://immutable-js.com

[2] https://clojure.org/about/rationale

Aufgabe 3: «Vier gewinnt» mit Undo (Miniprojekt)

Nun sollen die in Aufgabe 2 implementierten Funktionen verwendet werden, um eine neue Version des Spiels «Vier gewinnt» zu implementieren. Es soll nun eine Undo-Funktion erhalten, mit der im Spielverlauf Schritt für Schritt zurückgegangen werden kann, bis das Spielfeld wieder leer ist. Wenn wir dafür sorgen, dass das Zustandsobjekt nicht direkt verändert wird, können die Spielstände einfach in einem Array gespeichert werden. Dabei ist es kein Problem, wenn diese Objekte im Array gemeinsame Teilstrukturen haben, solange diese unverändert sind.

- Legen Sie zusätzlich zur globalen Variablen *state* eine weitere ebenfalls globale Variable *stateSeq* an, welche als leeres Array initialisiert wird. Dass wir hier mit globalen Variablen arbeiten, ist ein kleiner Schönheitsfehler der Implementierung, den wir mit State Hooks (Thema nächste Woche) beheben können.
- Alle Zustandsänderungen geschehen im Click-Event-Handler der Applikation. Passen Sie die Zustandsänderungen mit Hilfe der Funktionen *setInList* und *setInObj* so an, dass ein neues Zustandsobjekt erstellt wird, ohne den alten Zustand zu verändern.
- Direkt vor dem Ändern des Zustands muss der bisherige Zustand im Array gespeichert werden: `stateSeq.push(state)`
- Ergänzen Sie den HTML-Code durch einen Button mit der Klasse *undo*. Der zugehörige Event-Handler muss den letzten Zustand aus dem Array holen und das Board neu zeichnen.

Wenn alles stimmt, müsste es nun möglich sein, durch Klicken auf *Undo* Schritt für Schritt bis zum Ausgangspunkt des Spiels zurückzukehren.

Ergebnis: Wenn mit unveränderbaren Datenstrukturen gearbeitet wird, lässt sich eine Ergänzung wie die Undo-Funktion mit wenigen Zeilen Code umsetzen.