WMC Übung Tabellen in React Vite

Reactonly

Themen:

- Verschiedene Tabellen Packages: HTML, Material UI
- Klick Events auf Zellen
- Context

Ziele:

1. Erstelle ein React Vite Projekt.
2. Erstelle einen Context, der in alle Views verwendet werden kann.
3. Erstelle die View Komponenten HtmlTable.tsx und MUITable.tsx.
4. Erstelle ein Layout.tsx, um zu alle Views zu routen.
5. In der App.tsx erstelle entsprechende Routen mit einem Layout-Wrapper für die Navigation zu den Routen und einen ContextWrapper für den Zugriff auf Context Props.

Model - für alle Views gleich

1. **interface CellData** { row: number; column: string; color: string; besetzt: boolean; text: string; }

   Beispiel für cellData:

   Angenommen, du hast die Zellen 1-a, 2-b und 3-c gefüllt, dann könnte das cellData-Objekt so aussehen:

   ```
   { "1-a": { "row": 1, "column": "a", "color": "green", "besetzt": true, "text": "Zelle 1a (besetzt)" },

    "2-b": { "row": 2, "column": "b", "color": "red", "besetzt": true, "text": "Zelle 2b (besetzt)" },

    "3-c": { "row": 3, "column": "c", "color": "green", "besetzt": true, "text": "Zelle 3c (besetzt)" } }
   ```

2. **State cellData** um Inhalte der Zellen zu speichern
   - **const [cellData, setCellData] = useState**<{ [key: string]: CellData }>({});
   - **Key**: Der key für die Zellen im cellData-State besteht aus einer Kombination von **Zeile** und **Spalte**, um jede Zelle eindeutig zu identifizieren. Der Schlüssel wird als String im Format ${row}-${column} erstellt.
     Beispiel: Zeile 1, Spalte 'A' würde den Schlüssel "1-A" haben
     - const cellKey = `${row}-${column}`;

Programmierschritte – für alle Views gleich

Schritt 1: Erstelle eine LanguageContext, um Texte dynamisch in verschiedenen Sprachen in den View-Komponenten anzuzeigen.

Programmierung:

1. **interface LanguageContextType** { language: Language; texts: typeof texts['de']; // Dies zeigt entweder die deutschen oder englischen Texte setLanguage: (lang: Language) => void; }
2. **type Language = 'de' | 'en';** : Zur Definition der Texte für Deutsch und Englisch
   const texts = { de: { greeting: 'Hallo', selectColor: 'Wähle eine Farbe', green: 'Grün', red: 'Rot', clickCell: 'Klicke auf eine Zelle', occupied: 'ist besetzt. Wähle eine andere.', },
   en: { greeting: 'Hello', selectColor: 'Select a color', green: 'Green', red: 'Red', clickCell: 'Click on a cell', occupied: 'is occupied. Choose another.', }, };
3. **Texts Definition**: Enthält die Texte für die aktuelle Sprache, entweder Deutsch oder Englisch.
4. **State language**: Ermöglicht das Umschalten zwischen Deutsch (de) und Englisch (en).
5. **Dynamische Texte:** Alle Texte in der Komponente, wie z.B. der Titel für die Farbauswahl, die Farbe der Buttons oder die Nachricht, dass eine Zelle besetzt ist, basieren auf der aktuell ausgewählten Sprache.
6. **Buttons für Sprachauswahl:** Es gibt zwei Buttons, um zwischen Deutsch und Englisch umzuschalten.

Schritt 2: Für die React-Komponente in TypeScript mit Vite, erstelle eine 3x3 Tabelle mit Spaltenbeschriftungen Spalte A, Spalte B, Spalte C und Zeilenbeschriftungen Zeile 1, Zeile 2, Zeile 3 erstellt. Die Tabelle enthält ein Event, das es erlaubt, beim Klicken auf eine Zelle die Zeile und Spalte der Zelle zu erkennen und einen Text „Zeile und Spalte“ in diese Zelle zu schreiben.

Programmierung:

1. **Tabelle und Beschriftungen**: Die Tabelle hat eine Kopfzeile mit Spaltenbeschriftungen Spalte A, Spalte B, Spalte C und Zeilenbeschriftungen Zeile 1, Zeile 2, Zeile.
2. **Event-Handling**: Für jede Zelle gibt es ein onClick-Event, das die entsprechende Zeile und Spalte als Parameter an den handleClick-Handler übergib und der Text "Zelle 1-A (besetzt)“ direkt in die Zelle geschrieben.
   Hinweis: Die Spalten bzw. Zeilennummer wird mit einem String-Funktion aus den Spalten bzw. Zeilenbeschriftungen umgewandelt z.B. von „Spalte A“ , Zeile 1: 1-A umgewandelt.
   Hinweis: Es wird zuerst der Schlüssel berechnet, der eine Kombination aus Zeilen- und Spaltennummer ist.
3. **getCellText-Funktion**: Diese Funktion gibt den Text für eine Zelle zurück, falls vorhanden. Ansonsten wird ein leerer Text ““ zurückgegeben, wenn die Zelle noch nicht besetzt ist.
4. **Styling:** Die Zellen sind klickbar dank der CSS-Eigenschaft cursor: 'pointer'.

Schritt 3: Für die View-Komponente, erstelle zwei Buttons (Green und Red), um eine Farbe auszuwählen, und beim Klicken auf eine Zelle die gewählte Farbe anwendet.

Programmierung:

1. **State für Farbe**: selectedColor: Wird verwendet, um die aktuell ausgewählte Farbe (Grün oder Rot) zu speichern.
2. **State für Zellenfarben**: cellColors: Ein Objekt, das die Farben der einzelnen Zellen speichert, wobei der Schlüssel eine Kombination aus Zeilen- und Spaltennummer ist, z. B. 1-a, um die Farbe der Zelle eindeutig zu identifizieren.
3. **handleClick**: Beim Klicken auf eine Zelle wird zuerst der Schlüssel berechnet, die eine Kombination aus Zeilen- und Spaltennummer ist, z. B. 1-A ausgewählte Farbe auf diese Zelle angewendet, indem die Hintergrundfarbe der Zelle aktualisiert wird.
4. **handleColorSelection**: Dieser Handler setzt die selectedColor, je nachdem, welcher Button (Grün oder Rot) geklickt wird.
5. **getCellColor**: Diese Funktion gibt die Hintergrundfarbe für jede Zelle anhand vom Zellen-Key basierend auf dem gespeicherten Zustand zurück.

Schritt 4: Wenn eine Zelle schon besetzt ist durch Text, dann wird eine Popup - Message angezeigt. "Zelle <Spalte>- <Zeile> ist besetzt. Wähle eine andere Zelle.

Programmierung:

- **Warnung, wenn eine Zelle bereits besetzt ist**: Wenn auf eine bereits besetzte Zelle geklickt wird (d.h. wenn die Zelle bereits einen Text), wird eine Popup-Nachricht „Zelle <Z>-<S> besetzt!“ angezeigt.
  Wenn nur die Farbe fehlt, wird die Popup-Nachricht „Farbe fehlt für Zelle <Z>-<S>“.

Schritt 5: Beim Klick auf eine Zelle soll die Informationen Zeile, Spalte und Farbe und "besetzt":boolean" in einem Json-Objekt im LocalStorage gespeichert wird. Beim ersten Laden der Zeile wird der LocalStorage in der Tabelle dargestellt.

Programmierung:

1. **LocalStorage:** : Bei jedem Klick auf eine Zelle wird die cellData in einem JSON-Objekt gespeichert und im LocalStorage abgelegt.
2. **Init - Laden der Zelleninformationen aus dem LocalStorage**: Beim ersten Laden der Tabelle wird das LocalStorage ausgelesen und die gespeicherten Zelleninformationen auf die Tabelle angewendet.

Schritt 6: Für die View-Komponente, erstelle einen zusätzlichen Button „Edit“ und für die Auswahl einer Zelle die Listen mit Namen ZeilenListe (A,B,C) und ZeilenSpalten. Beim Klicken wird der Inhalt von der Zelle in einem Inputfeld auf eine Zelle die gewählte Farbe anwendet.

Programmierung:

1. **State für Farbe**: selectedColor: Wird verwendet, um die aktuell ausgewählte Farbe (Grün oder Rot) zu speichern.

State für Zellenfarben: cellColors: Ein Objekt, das die Farben der einzelnen Zellen

Views mit Tabellen

View HtmlTable.tsx

1. Kein **import** für HTML-Table von react notwendig
2. Durchlaufe alle Programmierschritte

View MuiTable.tsx

1. Verwende **import** { Box, IconButton, FormControlLabel, FormLabel, Radio, RadioGroup, Table, TableBody, TableCell, TableContainer, TableHead, TableRow, Paper, } from **'@mui/material';**
2. Durchlaufe alle Programmierschritte