React-Schulung

A. Einführung

1. Erstellen des Projekts (Link)

```
npx create-react-app {{Unser Projektname}} --template typescript
```

2. Zurechtfinden im Projekt

- Was wurde erstellt?

3. Erstmaliges starten des Projekts

```
npm run start
```

4. Allgemeine Fragen

Was ist eine React-Komponente?

- Eine React-Komponente ist im Grunde nur eine Funktion, welche HTML zurückgibt.

```
function App() {
      // Hier steht natives HTML oder eine andere React-Komponente
      return (
          <div className="App">
              <header className="App-header">
                  <img src={logo} className="App-logo" alt="logo" />
                  <p>Edit <code>src/App.tsx</code> and save to reload.</p>
                  <a className="App-link" href="https://reactjs.org"
target="_blank" rel="noopener noreferrer">
                  Learn React
                  </a>
              </header>
          </div>
      );
  }

  export default App;
```

Nutzen von Klassen als React-Komponente

```

```

```
class Hello extends React.Component<Props, object> {
    public render() {
        const { name, enthusiasmLevel = 1 } = this.props;

        if (enthusiasmLevel <= 0) {
            throw new Error('You could be a little more enthusiastic. :D');
        }

        return (
            <div className="hello">
                <div className="greeting">
                Hello {name + getExclamationMarks(enthusiasmLevel)}
                </div>
            </div>
        );
    }
}
```

Was sind Props?

- Props sind statische Informationen/Eigenschaften in einer Komponente, wie z.B. eine Refferenz zu einer Globalen Funktions-Klasse
- Props können in der Komponente während ihrem Lebenszyklus nicht verändert werden.

Was ist der State?

- State sind dynamische Informationen, welche sich während ihrem Lebenszyklus noch verändern können.
- Eine Veränderung am State sorgt dafür, dass sich die Komponente erneut rendert (nur die Teile, welche sich verändert haben).

B. Unser Projekt

Erstellen einer App, welche uns erlaubt Notizen zu speichern, bearbeiten und zu löschen.

1. Konzeption

- Wie soll unsere App aussehen?
- Wie sieht die Nutzerführung aus?

2. In welche Teile können wir unsere App unterteilen?

- Welche Fluent-UI-Controls können wir verwenden, um diese Funktion zu erzielen?
    - Action Button
    - Panel
    - Modal

- Dialog
- TextField
- Icons
- Buttons
- Spinner

3. Installieren der Abhängigkeiten

- Fluent-UI (Link)

```
npm install @fluentui/react --save
```

- JavaScript Cookie (Link)

```
npm i js-cookie @types/js-cookie --save
```

- Guid TypeScript (Link)

```
npm i guid-typescript --save
```

4. Wie erstelle ich mir meine eigene React-Komponente

```
class Name_der_Komponente extends
React.Component<Eigenschafts_Interface_der_Komponente,
State_Interface_der_Komponente> {

    construktor(props: Eigenschafts_Interface_der_Komponente){
        super(props);
        // Weiterer Setup und erstmaliges setzen des States
    }

    // Rendert die Komponente
    public render() {
        return (
        <div></div>
        );
    }
}
```

5. Wo und Wie wird die Komponente gerendert?

- Die Komponente wird in der index.tsx erstmalig gerendert. (Dies kann auch eine andere TS-Datei sein.)
- Wie wird die Komponente gerendert?

```
ReactDOM.render(<Name_der_Komponente />, document.getElementById('root'));
```

6. Nutzen von Properties

Beim Rendern einer Komponente können verschiedene Properties/Eigenschaften angegeben werden, diese entsprechen im größten Sinne den Attributen eines HTML-Tags. Ein großer Unterschied besteht darin, dass dort z.B. auch Methoden oder andere Datentypen übergeben werden können.

Es ist immer empfohlen eine strikte Typisierung einzuhalten, um unnötigen Bugs aus dem Weg zu gehen.

Properties können durch `this.props.EIGENSCHFTSNAME` abgerufen werden.

!! Properties können nicht in der Komponente verändert werden, sondern müssen von einer *Parent-Komponente* verändert werden.

Wann sind Properties hilfreich und wann nicht?

7. Nutzen des States

Um Daten *Live* zu ändern (z.B. bei einer Eingabe eines Users) wird häufig der sogenannte State verwendet. Dieser erlaubt es nur veränderte Teile neu zu rendern, jedoch auch nicht zu oft, damit die Performance nicht beeinträchtigt wird.

Der State kann durch `this.state.EIGENSCHFTSNAME` abgerufen werden.

`this.state` ist eine Konstante und kann daher nicht mit `this.state = DEIN_NEUER_STATE;` verändert werden. Um den State zu verändern rufen wir die Funktion `this.setState({EIGENSCHFTSNAME: NEUER EIGENSCHFTSWERT});` auf.

Dabei können wir sowohl nur eine Eigenschaft als auch mehrere Eigenschaften geichzeitig verändern.

C. Events

Die Behandlung von Ereignissen mit React-Elementen ist mit der Behandlung von Ereignissen auf DOM-Elementen vergleichbar.

In HTML:

```
<button onclick="clickEvent()">
  Klick mich!!
</button>
```

In React:

```
<button onClick={clickEvent.bind(this)}>
  Klick mich!!
```

```
</button>
```

Wobei `clickEvent` eine funktion in der TSX oder einer anderen Datei ist:

```
function clickEvent(){
    console.log("clicked");
}
```

Um Änderungen am State vorehmen zu können, die Funktion mit `FUNKTIONSNAME.bind(this)` übergeben werden. Dabei werden die Übergabeparameter automatisch an die neue Funktion weitergeleitet.

Folgende Funktionen erzielen die gleiche Funktion:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

D. Usefull Tipps

1. "*Mappen*" von Arrays

Um z.B. eine Liste mit verschiedenen Items zu rendern, kann man in einer Funktion ein Array *mappen*.

Wichtig danei ist, dass das Element, was zurückgegeben wird ein **key-Attribut** besitzt, da React ansonsten das falsche Element erneut rendert, wenn z.B. ein Element am Anfang hinzukommt. Dabei ist es auch nicht empfohlen, den Index als key zu wählen, da dieser nicht bei jedem Element eindeutig ist.

```
public render() {
    let items: string[] = ["Apfel", "Banane", "Kiwi", "Orange"];
    return (
        <ul>
            {
                items.map((item: string, index: number, array: typeof string[])=>{
                    return (
                        <li key={SOME_UNIKE_ITEM_PROPERTY_OR_GUID}>{item}</li>
                    );
                })
            }
        </ul>
    );
}
```

Expected output:

```
<ul><li>Apfel</li><li>Banane</li><li>Kiwi</li><li>Orange</li></ul>
```

2. Konditionales Rendern von Komponenten

Wenn man in einer Komponente Teile nur rendern möchte, wenn diese eine bestimmt Bedingung erfüllen, z.B. Wenn ein Textfeld nicht leer ist.

Damit folgendes Element nicht gerendert wird, kann die folgende Struktur verwedet werden:

```
public render() {
    return (
        <div>
            {true &&
                <p>Dieser Paragraph wird angezeigt</p>
            }
            {false &&
                <p>Dieser Paragraph wird nicht angezeigt</p>
            }
        </div>
    );
}
```

Diese Bedingungen können dann auch verknüpft werden.

```
public render() {
    let name: string = "Dein Name";
    return (
        <div>
            {name.length > 0 && name.length < 10 &&
                <p>Dein Name ist {name}</p>
            }
            {name.length == 0 || name.length > 10 &&
                <div>Der Name {name} ist leer oder zu lang.</div>
            }
        </div>
    );
}
```

3. Nützliche Funktionen

componentDidMount()

Um eine bestimmte Aktion auszuführen, wenn die Komponente auf der Seite platziert wurde (z.B. Daten aggregieren) kann die Methode `componentDidMount()` verwendet werden. Darin kan dann `this.setState({});` verwendet werden, um ein erneutes Rendern zu triggern und die Komponente zu verändern.

```
/**
 * Called immediately after a component is mounted. Setting state here will
trigger re-rendering.
 */
componentDidMount?(): void;
```

shouldComponentUpdate()

Um eine das erneute Rendern zu unterdrücken, z.B. um ein Focus Event beizubehallten, kann die Methode `shouldComponentUpdate()` verwendet werden diese gibt dann einen Boolean (true/false) zurück.

- true = Komponente soll erneut gerendert werden
- flase = Komponente soll nicht erneut gerendert werden

Die Methode hat drei Parameter:

1. nextProps

- Enthalten die neuen Properties / Eigenschaften.
- Können mit `this.props` verglichen werden.

2. nextState

- Der neue State, falls die Methode durch `this.setState({});` getriggert wurde.
- Kann mit `this.state` verglichen werden um zu entscheiden, ob die Komponente ein Update erhalten sollte.

3. nextContext

- Wird global verwendet um die Eigenschaften nicht durch den gesamten Komponenten Aufbau zu schleifen.

```
/**
 * Called to determine whether the change in props and state should trigger a re-
render.
 *
 * `Component` always returns true.
 * `PureComponent` implements a shallow comparison on props and state and returns
true if any
 * props or states have changed.
 *
 * If false is returned, `Component#render`, `componentWillUpdate`
 * and `componentDidUpdate` will not be called.
 */
shouldComponentUpdate?(nextProps: Readonly<P>, nextState: Readonly<S>,
nextContext: any): boolean;
```

componentWillUnmount()

Wird die Komponente entfernt, können hier notwendige Aufräumarbeiten angestoßen werden, z.B. ein GET/POST/PUT Request abgebrochen oder ein Timeout / Intervall gecleared.

```
/**
 * Called immediately before a component is destroyed. Perform any necessary
cleanup in this method, such as
 * cancelled network requests, or cleaning up any DOM elements created in
`componentDidMount`.
 */
componentWillUnmount?(): void;
```

componentDidCatch()

Um einen Fehler beim Rendern abzufangen wird die Methode `componentDidCatch()` implementiert.

Fehler können z.B. sein, dass eine Variable `undefined` oder `null` ist.

Der Übergabeparameter `error: Error` enthält den Stack des Fehlers.

Der Übergabeparameter `errorInfo: ErrorInfo` enthät eine Property Namens `componentStack`, welche erfasst, welche Komponente die Ausnahme enthielt sowie Ihren callStack.

Hier könnte z.B. dem Nutzer eine Nachricht angezeigt werden, welcher Fehler aufgetreten ist, und wie dieser behoben werden könnte oder die Eigenschaften auf Fehler überprüft und die Komponente dann erneut gerendert werden.

```
/**
 * Catches exceptions generated in descendant components. Unhandled exceptions
will cause
 * the entire component tree to unmount.
 */
componentDidCatch?(error: Error, errorInfo: ErrorInfo): void;
```

E. Das fertige Projekt

1. Aufbau

```
│   .gitignore
│   package-lock.json
│   package.json
│   README.md
│   tsconfig.json
│
├──public
│       favicon.ico
│       index.html
```

```
|       manifest.json
|       robots.txt
|
└──src
    |   functions.ts
    |   index.css
    |   index.tsx
    |   interfaces.ts
    |   react-app-env.d.ts
    |   serviceWorker.ts
    |   setupTests.ts
    |
    └──components
            Note.tsx
            Notizanzeige.tsx
            Notizbearbeitung.tsx
```

Im `src` Ordener befindet sich unsere Lösung.

In der Datei `functions.ts` befinden sich die allgemeinen Funktionen `getNotes()` und `saveNotes()`, welche uns helfen unseren notize abzurufen und zu speichern.

In der Datei `index.tsx` befindet sich der Einstiegspunkt für unser Programm.

```
ReactDOM.render(
    <Notizanzeige />,
  document.getElementById('root')
);
```

Diese Zeile rendert unsere erste Komponente: Die Notizanzeige. Dabei wird der Inhalt des HTML-Elements, mit der Id `root`, mit dem Inhalt unserer Komponente ersetzt.

2. Beschreiben unserer Komponenten

Notizanzeige

Unsere Komponente `<Notizanzeige />` beinhaltet die Anzeige für die erstellten Notizen.

```
<>
    {/* Button wecher dafür verwendet wird neue Notizen hinzuzufügen */}
    <ActionButton iconProps={{ iconName: "Add" }} text={"Notiz hinzufügen"}
onClick={this.addNote.bind(this)} />
    {/* Dier Lassen wir uns einen Dialog anzeigen, welcher uns eine neue Notiz
hinzufügt */}
    {this.state.newNote !== null &&
        <Notizbearbeitung Note={this.state.newNote} onDelete=
{this.discardNewNote.bind(this)} onDiscard={this.discardNewNote.bind(this)}
onSave={this.appendNote.bind(this)} />
    }
```

```
    {/* Dieser Abschnitt wird angezeigt, wenn Notizen vorhanden sind, und
bestätigt wurde, dass derzeitig keine Notizen geladen wurden */}
    {this.state.notes.length > 0 && !this.state.loading &&
        // Hier werden alle Notizen einzeln gerendert
        this.state.notes.map((value: iNote, index: number, array: iNote[]) => {
        return (
            <div key={value.guid}>
                <Note Note={value} onSave={this.saveNote.bind(this, index)}
onDelete={this.deleteNote.bind(this, index)} />
                <hr />
            </div>
        );
        })
    }
    {/* Dieser Abschnitt wird angezeigt, wenn keine Notizen vorhanden sind, und
bestätigt wurde, dass derzeitig keine Notizen geladen wurden */}
    {this.state.notes.length === 0 && !this.state.loading &&
        <MessageBar messageBarType={MessageBarType.info}>Du hast keine
Notizen</MessageBar>
    }
    {/* Dieser Abschnitt wird angezeigt, wenn derzeit keine Notizen geladen werden
*/}
    {this.state.loading &&
        <Spinner size={SpinnerSize.large} type={SpinnerType.large} label={"Lade
Notizen"}></Spinner>
    }
</>
```

Dabei werden verschiedene Events handgehabt:

1. Event beim Laden der Komponente

Um die Notizen automatisch beim Laden der Komponente abzufragen wird in der Methode `componentDidMount()` die Funktion zum Abfragen der Notiz aufgerufen und der State der Komponente dementsprechend gesetzt.

```
public componentDidMount() {
    NoteFunctions.getNotes()
        .then((value: iNote[]) => {
        this.setState({ notes: value, loading: false });
        });
}
```

2. Events beim Hinzufügen einer Notiz:

```
private addNote() {
    this.setState({ newNote: { body: "", title: "", guid: Guid.create().toString()
} });
}
```

```
private appendNote(newNote: iNote) {
    let newNotes: iNote[] = this.state.notes;
    newNotes.unshift(newNote);
    this.setState({ notes: newNotes, newNote: null });
    NoteFunctions.saveNotes(newNotes);
}

private discardNewNote() {
    this.setState({ newNote: null });
}
```

3. Events beim Speichern oder entfernen einer Notiz:

```
private saveNote(index: number, newNote: iNote) {
    let newNotes: iNote[] = this.state.notes;
    newNotes[index] = newNote;
    this.setState({ notes: newNotes });
    NoteFunctions.saveNotes(newNotes);
}

private deleteNote(index: number) {
    let newNotes: iNote[] = this.state.notes;
    newNotes.splice(index, 1);
    this.setState({ notes: newNotes });
    NoteFunctions.saveNotes(newNotes);
}
```

Notiz

Dise Komponente stellt eine Notiz dar und erlaubt es dem Nutzer genau diese eine zu bearbeiten.

```
<>
    <p>{this.props.Note.title}</p>
    <pre>{this.props.Note.body}</pre>
    <ActionButton text={`Bearbeiten`} iconProps={{ iconName: "Edit" }} onClick=
{this.toggleEditMode.bind(this)} />
    <ActionButton text={`Löschen`} iconProps={{ iconName: "Delete" }} onClick=
{this.props.onDelete} />
    {this.state.EditModeOpen &&
        <Notizbearbeitung Note={this.props.Note} onSave={this.saveNote.bind(this)}
onDelete={this.props.onDelete} onDiscard={this.toggleEditMode.bind(this)} />
    }
</>
```

Der Abschnitt `<p>{this.props.Note.title}</p> <pre>{this.props.Note.body}</pre>` stellt den Titel sowie den Body der Notiz dar. Da der Body auch mehrere zeilen beinhlatet, wird dafür der HTML-Tag `<pre>`

</pre> verwendet, welcher leere auch "*leere*" Zeichen (wie Leerzeichen oder newLinkeCharakters) darstellt.

Der Bearbeitungsdialog wird Konditional gerendert, damit dieser durch einen Klick auf einen Button angezeigt werden kann. Dazu wurde folgende Funktion erstellt.

```
private toggleEditMode() {
    this.setState({ EditModeOpen: !this.state.EditModeOpen });
}
```

Diese setzt einen State-Eigenschafts-Wert und triggert anschließend ein erneutes Rendern.

Notizbearbeitung

Diese Komponente stellt unseren Bearbeitungsdialog dar, in welchem wir unseren Titel und Body bearbeiten sowie die Notiz löschen können.

Die Komponente ist so aufgebaut:

```
<>
    <Dialog
        hidden={false}
        onDismiss={this.saveNote.bind(this)}
        dialogContentProps={{
            type: DialogType.normal,
            title: 'Bearbeiten',
            showCloseButton: true,
            onDismiss: this.props.onDiscard.bind(this)
        }}
    >
        <TextField
            label={"Titel eingeben:"}
            value={this.state.Title}
            onChange={this.titleChanged.bind(this)}
        />
        <TextField
            label={"Body eingeben:"}
            value={this.state.Body}
            multiline
            autoAdjustHeight
            resizable={false}
            onChange={this.bodyChanged.bind(this)}
        />
        <DialogFooter>
            <PrimaryButton onClick={this.saveNote.bind(this)} text={"Speichern"}
/>
            <DefaultButton onClick={this.props.onDelete.bind(this)} text=
{"Löschen"} />
        </DialogFooter>
    </Dialog>
</>
```

Hier werden im Dialog zwei Textfelder erstellt, welche den State nutzen, um ihre Werte anzuzeigen und zu ändern. Dabei wird nicht auf das *interne* Statemanagement gesetzt, sondern auf eine eigene Funktionen sesetzt, welche uns den State setzen.

Diese sind wie folgt aufgebaut:

```
private titleChanged(event: React.FormEvent<HTMLInputElement |
HTMLTextAreaElement>, newTitle: string) {
    this.setState({ Title: newTitle });
}

private bodyChanged(event: React.FormEvent<HTMLInputElement |
HTMLTextAreaElement>, newBody: string) {
    this.setState({ Body: newBody });
}
```

Die Funktion `titleChanged()` setzt den `Title` Wert des States neu, wohingegen die Funktion `bodyChanged()` den `Body` Wert des States neu setzt. Beide Funktionen triggern ein erneutes rendern des jeweiligen Control, wodurch der neue Wert übernommen wird.

Hier ist auch zu erkennen, dass Übergabeparameter mit dem Funktionszusatz `.bind(this)` automatisch weitere Parameter wie das Event oder den neuen Wert zugewiesen bekommen.

Der Dialog erlaubt es einem, einen Footer zu erstellen, welcher am Ende des Dialogs angezeigt wird.