# Udemy Ecommerce React

## React Concepts

1. Don’t touch the DOM. I‘ll do it!
2. Build webistes like lego blocks
3. Unidirectional dataflow : Ermögllicht leichtes debuggen, weil es eben nur eine Stelle gibt, wenn ein Fehler passiert. Nämlich dort, wo die fehlerhaften Daten festgehalten sind
4. I’m just UI

**Imperatives Vorgehen** war früher = DOM Manipulation. Performance-hungrig

**Deklaratives Vorgehen** = REACT. KEINE DOM Manipulation. React kümmert sich darum, wie Daten ausschauen sollen. Es weiß bereits selbst, wo was passieren muss in einem bestimmten Fall. Deshalb „react“ auf die Daten, die ich dir gebe

Komponenten sind im Endeffekt nur Javascript Funktionen bzw. Klassen

## Kernprinzipien, auf die es ankommt

1. Decide on Components (wieviele)
2. Decide the State and where it lives
3. What changes when state changes

## Schnellstart React mit “Create React App”

Zuerst braucht es generell NODE.JS und dann am besten NODE VERSION MANAGER. Damit man NODE einfacher updaten kann mit **nvm install 18.2.2**

Dann schreibt man **nvm use 18.2.2**

Anschließend **npx create-react-app NAME\_DER\_APPLIKATION**, wodurch das Webpack und Babel und alle möglichen Dateien konfiguriert werden

In **package.json** gibt es wieder **scripts**, die man einfach mit **npm SCRIPT\_NAME** anfahren kann. **Npm start** oder **npm build** (erstellt einen neuen **build** Ordner. Diese Daten werden dann deployed)

## Babel

„Konvertiert“ Javascript immer in die für den Browser verständliche Version

## Webpack

Nimmt alle Javascript-Dateien und „minimiert“ bzw. optimiert.

## JSX (nur innerhalb des returns des HTML Codes)

JSX ist wie in der Überschrift festgehalten, nur innerhalb des return Bereichs von render, wo HTML steht. Also immer, wenn man einen neuen HTML Block returned, zB innerhalb eines array.**map** Aufrufs, wo man für jeden Eintrag HTML Code schreibt, muss man den Code wieder in die curly braces setzen

## Unique Keys für Elemente

Returnte HTML-Elemente innerhalb unseres JSX MÜSSEN **key** Attribute haben

## LifeCycleMethods laufen, wenn Komponente gerendered wird

zB **ComponentDidMount(){CODE}** wird immer angefahren, wenn eine Komponenten gerendered wird

## Promises Async Await Functions

ACHTUNG NODE.JS kennt fetch nicht von Haus aus!

fetch("Testurl.com")

   .then((response) => console.log(response))

   .catch((error) => console.log("I failed"));

**fetch** returned einen **new Promise()**. Deshalb muss man nicht selbst **const promise = new Promise(„URL“)** schreiben. Resolve und Reject können praktisch sein, um zu überprüfen, um welche Daten es sich handeln soll, die man erhalten hat. Möchte man zB eine Zahl und bekommt einen String, dann rejecte

*const* myPromise = new Promise((resolve, reject) => {

*if* (resolve) {

      console.log("Got something");

   } *else* {

      console.log("I failed again");

   }

});

### Async Await

Mit dem **async() => {**await**}** weiß Javascript, dass wir eine asynchrone Funktion haben wollen. Dadurch wird immer gewartet.

Der Vorteil ist, dass der Code wie synchroner Code lesbar ist. Im Endeffekt eh wie **.then/catch-chaining**. Es geht nur darum, was besser lesbar ist. Das Error Handling funktioniert hier mit **try/catch**

*const* myAsyncFunction = *async* () => {

*try* {

*const* usersResponse = *await* fetch(

         "https://jsonplaceholder.typicode.com/users"

      );

*const* users = *await* usersResponse.json();

*const* firstUser = users[0];

*const* postsResponse = *await* fetch(

         "https://jsonplaceholder.typicode.com/posts?userId=" + firstUser.id

      );

*const* posts = *await* postsResponse.json();

      console.log(posts);

   } *catch* (err) {

      console.log("There was an error");

   }

};

## Folder structure

Man kann zB die Komponenten Dateien auch mit **.jsx** enden lassen, wenn darin JSX vorkommt. Javascript wird trotzdem normal erkannt. Das muss man aber nicht, aber es hilft wieder für eine weitere Unterteilung!

Weiters zahlt es sich aus, dass man innerhalb des **components** Ordner die Dateien mit **name.component.jsx** benennt, weil so weiß man gleich wieder, dass es sich um die Komponente handelt und bei **name.styles.css** um die Styling Datei

## Props und Children

Props übergibt man, indem ein Attribut setzt.

<CardList *name*="Paul" />

Man kann eine Komponente auch mit einem separaten closing Tag versehen, wodurch der Inhalt darin automatisch ein props.children wird

<CardList *name*="Paul">

               <h1>Paul</h1>

            </CardList>

Während **props.name** einfach nur als Text innerhalb des divs gerendered werden würde, stünde **props.children** innerhalb einer h1

*export* *const* CardList = (props) => {

   console.log(props);

*return* (

      <div>

         Hello {props.name} & {props.children}

      </div>

   );

};

## Arrow-Function-Komponente mit importiertem stylesheert in App-Komponente übergeben

**App-Komponente**

Wir übergeben hier der CardList-Komponente children

   render() {

*return* (

         <div *className*="App">

            <CardList>

               {*this*.state.monsters.map((monster) => (

                  <h1 *key*={monster.id}>{monster.name}</h1>

               ))}

            </CardList>

         </div>

      );

   }

**Arrow-Function-Komponente mit importiertem Stylesheet**

Wir geben hier dem container-div die Klasse aus dem importierten Stylesheet. props.children sind die gemappeden **H1**

*import* React *from* "react";

*import* "./card-list.styles.css";

*export* *const* CardList = (props) => {

*return* <div *className*="card-list">{props.children}</div>;

};

**CSS**

*.card-list* {

   width: 85*vw*;

   margin: 0 auto;

   display: grid;

   grid-template-columns: repeat(4, 1*fr*);

   grid-gap: 20*px*;

}

## String Interpolation mit Backticks

Im Projekt wird davon geredet für die **src** von **img**, weil nur mit **backticks** der string interpoliert werden kann

Für die Monster-Bilder brauchen wir unterschiedliche Zahlen in der URL, die wir durch die monster.id bekommen. Aber den src String muss man dann eben mit backticks schreiben und mit **${}** den Wert einsetzen

*src*={`https://robohash.org/${props.monster.id}?set=set2&size=180x180`}

## Gegenwärtiger Stand

Eine CardList-Komponente, welche im renderer über ein Array aus Daten mapped, wo für jede „Date“ eine separate Komponente erstellt wird

*export* *const* CardList = (props) => (

   <div *className*="card-list">

      {props.monsters.map((monster) => (

         <Card *key*={monster.id} *monster*={monster} />

      ))}

   </div>

);

*export* *const* Card = (props) => (

   <div>

      <h1>{props.monster.name}</h1>

   </div>

);

## setState() ist asynchron | Zweiter Parameter „umgeht“ dieses Verhalten

Würde man console.log einfach nach this.setState schreiben, dann wäre der Input beim 1.Tastendruck leer in der console

<input

*type*="search"

*placeholder*="search monster"

*onChange*={(e) => {

*this*.setState({ searchField: e.target.value }, () =>

                     console.log(*this*.state)

                  );

               }}

Möchte man also erst den Input abwarten, dann muss man den 2.Parameter verwenden

## Destructuring Objects

Man muss die Eigenschaftsnamen wissen, dann kann man sich einfach neue Variablen erstellen mit den Eigenschaftsnamen.

const objekt = {eigenschaft1: „Hey“, eigenschaft2: „du“}

Mit const/let {eigenschaft1, eigenschaft2} = objekt

Anschließend hat man die variablen eigenschaft1 und eigenschaft2 mit den Werten aus dem Objekt

*const* { monsters, searchField } = *this*.state;

*const* filteredMonsters = monsters.filter((monster) =>

         monster.name.toLowerCase().includes(searchField.toLowerCase())

      );

## Destructuring und state Weitergabe

Unsere SearchBox-Komponente bekommt 2 props, siehe Screenshot. Die 2.prop ist eine Callback Funktion, der die setState Methode mitgegeben wird mit dem Objekt searchfield, das den Wert erhalten soll, den der User eingibt. Das geniale daran ist, dass wir dadurch sie Searchbox immer wieder an unterschiedlichen Stellen mit unterschiedlichen placeholdern und Funktion oder sonstigen parametern versehen können!

<SearchBox

*placeholder*="search monsters"

*handleChange*={(e) => *this*.setState({ searchField: e.target.value })}

            />

*export* *const* SearchBox = ({ placeholder, handleChange }) => (

   <div>

      <input

*className*="search"

*type*="search"

*placeholder*={placeholder}

*onChange*={handleChange}

      />

   </div>

);

## Methoden innerhalb einer Klasse bind und arrow functions lexical scoping

Normalerweise muss man immer mit **.bind(this)** arbeiten, wenn man eine Funktion aus einer Klasse woanders hin übergibt.

**Arrow Functions LEXICAL SCOPING** binden **this** immer an das, wo sie definiert wurden. Das heißt, man muss dann nicht nochmal **.bind(this)** schreiben, da sich ihr **this** immer auf die „äußere/darüberliegende“ Klasse bezieht, wo sie definiert wurden. Könnte natürlich Probleme machen, wenn man 2 Ebenen tiefer geht oder?

Ebenfalls interessant ist, dass man innerhalb von Klassen NICHT let/var/const voranschreiben muss!

   handleChange = (e) =>

*this*.setState({

         searchField: e.target.value,

      });

Übergabe ohne **bind**

<SearchBox

*placeholder*="search monsters"

*handleChange*={*this*.handleChange}

            />