18 - React: Die Grundlagen - Komponenten, JSX & Props

Einleitung

- **Themen**: Dieses Skript markiert den Beginn unserer Reise in die moderne Frontend-Entwicklung mit React. Wir legen das Fundament, auf dem alle zukünftigen Interaktionen mit unserer Django-API aufbauen werden.
- **Fokus**: Ein tiefes, grundlegendes Verständnis der Kernkonzepte von React. Wir klären, *was* React ist und *warum* es so populär ist. Die drei Säulen von React – **Komponenten, JSX und Props** – werden detailliert und mit vielen Beispielen eingeführt.
- **Lernziele**:
  - Verstehen, was eine Single-Page Application (SPA) ist und welche Vorteile der komponentenbasierte Ansatz von React bietet.
  - Ein komplettes React-Projekt mit dem modernen Build-Tool `Vite` von Grund auf einrichten können.
  - JSX-Syntax sicher anwenden und den Unterschied zu reinem HTML kennen.
  - Eigene, wiederverwendbare "Function Components" erstellen.
  - Daten mit "Props" von übergeordneten zu untergeordneten Komponenten weitergeben, um dynamische UIs zu erstellen.

1. Was ist React und warum verwenden wir es?

Bisher hat unser Django-Server bei jedem Klick eine komplett neue HTML-Seite an den Browser geschickt. Moderne Webanwendungen fühlen sich oft eher an wie Desktop-Programme – schnell, flüssig und ohne ständiges Neuladen. Dies wird durch JavaScript-Bibliotheken wie React erreicht.

- **React ist eine JavaScript-Bibliothek zum Aufbau von Benutzeroberflächen (User Interfaces).**
  - Es ist nicht ein komplettes Framework wie Django, sondern kümmert sich nur um die "View"-Schicht – also alles, was der Benutzer sieht und womit er interagiert.
- **Deklarativer Ansatz**:
  - Anstatt dem Browser Schritt für Schritt zu sagen, *wie* er die Seite verändern soll (imperativ), beschreiben wir mit React nur, *wie die UI für einen bestimmten Zustand aussehen soll* (deklarativ). React kümmert sich dann darum, den DOM effizient zu aktualisieren. Das macht den Code lesbarer und weniger fehleranfällig.
- **Komponentenbasierte Architektur**:
  - Das Herzstück von React. Eine komplexe Benutzeroberfläche wird in kleine, unabhängige und wiederverwendbare Bausteine zerlegt, die **Komponenten** genannt werden.
  - **Analogie**: Stell dir eine Webseite wie ein Lego-Haus vor. Die Navigationsleiste ist ein Lego-Stein, jeder Button ist ein Stein, eine Produktkarte ist ein Stein. Man baut jeden Stein einmal und kann ihn dann beliebig oft wiederverwenden.
- **Single-Page Application (SPA)**:
  - React wird meistens verwendet, um SPAs zu erstellen. Das bedeutet: Der Server lädt anfangs nur eine einzige, fast leere `index.html`-Datei.
  - Danach übernimmt React die Kontrolle. Es "malt" die gesamte Anwendung mit JavaScript in diese eine Seite. Wenn der Benutzer navigiert, lädt React nicht die ganze Seite neu, sondern tauscht nur die notwendigen Komponenten aus. Das Ergebnis ist eine deutlich schnellere und flüssigere Benutzererfahrung.

2. Das erste React-Projekt mit Vite einrichten

Um mit React zu entwickeln, benötigen wir eine lokale Entwicklungsumgebung. Wir verwenden `Vite`, ein extrem schnelles und modernes Tool, das uns alles Nötige einrichtet.

Voraussetzung: Node.js muss auf deinem System installiert sein, da es `npm` (Node Package Manager) enthält, mit dem wir Vite und andere Pakete verwalten. Lade es von der offiziellen Node.js-Webseite herunter und installiere es.

Schritt 1: Projekt erstellen

Öffne ein Terminal (nicht im Django-Projekt, sondern in einem übergeordneten Ordner) und gib den folgenden Befehl ein:

```
```

```
# Dieser Befehl startet den interaktiven Setup-Prozess von Vite
npm create vite@latest
```

Vite wird dir nun einige Fragen stellen:

1. **Project name:** Gib einen Namen für dein Frontend-Projekt ein, z.B. `recipe-frontend`.
2. **Select a framework:** Wähle mit den Pfeiltasten `React` aus und drücke Enter.
3. **Select a variant:** Wähle `JavaScript` aus und drücke Enter.

Vite erstellt nun einen neuen Ordner `recipe-frontend` mit allen notwendigen Dateien.

Schritt 2: Abhängigkeiten installieren

Navigiere in den neuen Ordner und installiere die Projektabhängigkeiten.

```
# In den neuen Projektordner wechseln
cd recipe-frontend

# Alle in package.json definierten Pakete (wie react, react-dom) installieren
npm install
```

Schritt 3: Entwicklungsserver starten

Starte den lokalen Entwicklungsserver.

```
npm run dev
```

Dein Terminal zeigt dir nun eine lokale Adresse an, meist `http://localhost:5173/`. Öffne diese Adresse in deinem Browser. Du solltest die React-Startseite sehen. Das Besondere am Vite-Dev-Server ist **Hot Module Replacement (HMR)**: Wenn du eine Datei speicherst, wird die Änderung im Browser sofort sichtbar, ohne dass die Seite neu geladen wird.

Schritt 4: Projektstruktur verstehen

Öffne den Projektordner in deinem Code-Editor. Die wichtigsten Dateien und Ordner sind:

- `index.html`: Die einzige HTML-Datei deiner SPA. Der gesamte Inhalt wird von React in das `<div id="root"></div>` in dieser Datei eingefügt.
- `src/`: Der Ordner, in dem wir 99% unserer Zeit verbringen werden. Hier liegt unser gesamter React-Code.
- `src/main.jsx`: Der Einstiegspunkt der Anwendung. Hier wird die Haupt-React-Komponente (`App`) genommen und in das `root`-Div in der `index.html` "gemalt".
- `src/App.jsx`: Die Wurzel-Komponente deiner Anwendung. Von hier aus bauen wir unsere UI auf.

3. Exkurs: `import` und `export` in JavaScript (ES6-Module)

Um unseren Code zu organisieren, teilen wir ihn in separate Dateien, sogenannte **Module**, auf. Das `import`/`export`-System ist der Mechanismus, um Funktionen, Variablen oder Klassen aus einer Datei in einer anderen verfügbar zu machen. Dies ist für React absolut essenziell.

Es gibt zwei Hauptarten von Exports: **Named Exports** und **Default Exports**.

- **Named Exports (Benannte Exporte)**
    - Man kann beliebig viele benannte Exporte aus einer Datei haben.
    - Sie werden verwendet, um eine Sammlung von Hilfsfunktionen oder Konstanten zu exportieren.

  Beispiel `src/utils/math.js`:

  ```
  ```

```
// src/utils/math.js
export const PI = 3.14159;

export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}
```

Importieren von Named Exports: Der Import erfolgt mit geschweiften Klammern `{}`. Die Namen innerhalb der Klammern **müssen exakt** mit den exportierten Namen übereinstimmen.

```
// In einer anderen Datei, z.B. App.jsx
import { add, PI } from './utils/math.js';

console.log(`Der Wert von Pi ist ${PI}`);
console.log(`2 + 3 = ${add(2, 3)}`);
```

- **Default Export (Standard-Export)**
  - Jede Datei kann **nur einen** Default Export haben.
  - Er wird für das "Haupt-Ding" verwendet, das eine Datei bereitstellt – in React ist das typischerweise die Komponente selbst.

Beispiel `src/components/WelcomeMessage.jsx`:

```
// src/components/WelcomeMessage.jsx
function WelcomeMessage() {
  return <h2>Willkommen!</h2>;
}

// Exportiert die Funktion als Standard
export default WelcomeMessage;
```

Importieren eines Default Exports: Der Import erfolgt **ohne** geschweifte Klammern. Man kann dem Import einen beliebigen Namen geben, obwohl es üblich ist, den Namen der Komponente zu verwenden.

```
// In einer anderen Datei, z.B. App.jsx
import Greeter from './components/WelcomeMessage.jsx'; // Name "Greeter" ist frei wählbar

function App() {
  return <Greeter />;
}
```

4. Kernkonzept 1: JSX - HTML in JavaScript schreiben

JSX (JavaScript XML) ist eine Syntaxerweiterung für JavaScript. Sie erlaubt uns, HTML-ähnlichen Code direkt in unseren JavaScript-Dateien zu schreiben.

- **Warum JSX?** Es fühlt sich natürlich an, die Struktur einer Komponente direkt mit ihrer Logik zu verbinden.
- **Wichtig**: Browser verstehen kein JSX! Unser Build-Tool (Vite) verwendet im Hintergrund einen **Transpiler** (wie Babel), der unseren JSX-Code in normale JavaScript-Funktionsaufrufe (`React.createElement(...)`) umwandelt, bevor er an den Browser gesendet wird.

Beispiel für eine einfache Komponente mit JSX:

```
```

```jsx
// src/App.jsx

function App() {
  const userName = "Anna";
  // Das, was wie HTML aussieht, ist JSX
  return (
    <div>
      <h1>Hallo, {userName}!</h1>
      <p>Willkommen zu unserer ersten React-Anwendung.</p>
    </div>
  );
}

export default App;
```

Regeln für JSX:

1. **Ein einzelnes Wurzelelement**: Eine Komponente muss immer genau *ein* übergeordnetes Element zurückgeben. Wenn man kein zusätzliches `<div>` möchte, kann man einen "Fragment" verwenden: `<> ... </>`.
2. **JavaScript in geschweiften Klammern `{}`**: Man kann beliebige JavaScript-Ausdrücke (Variablen, Berechnungen, Funktionsaufrufe) direkt in JSX einbetten, indem man sie in `{}` setzt.
3. **Attribute wie in HTML, aber mit Unterschieden**:
   - `class` wird zu `className` (da `class` ein reserviertes Wort in JavaScript ist).
   - `for` (bei `<label>`) wird zu `htmlFor`.
   - Attribute werden in `camelCase` geschrieben (z.B. `onclick` -> `onClick`).
4. **Tags müssen geschlossen werden**: Jeder Tag muss entweder einen schließenden Tag haben (`<p></p>`) oder selbstschließend sein (`<img />`, `<br />`).

5. Kernkonzept 2: Komponenten - Die Bausteine der UI

Eine Komponente ist eine unabhängige, wiederverwendbare Einheit der Benutzeroberfläche. In modernem React ist dies einfach eine JavaScript-Funktion, die JSX zurückgibt.

Regeln für Komponenten:

- Der Funktionsname muss immer mit einem **Großbuchstaben** beginnen. Daran erkennt React, dass es sich um eine Komponente handelt und nicht um einen normalen HTML-Tag.

Beispiel: Erstellen und Verwenden von Komponenten

1. **Erstelle eine neue Komponente `RecipeCard.jsx`**: Erstelle einen neuen Ordner `src/components/` und darin eine neue Datei `RecipeCard.jsx`.

   ```jsx
   // src/components/RecipeCard.jsx

   function RecipeCard() {
     return (
       <div className="recipe-card">
         <h2>Spaghetti Carbonara</h2>
         <p>Ein klassisches italienisches Gericht.</p>
       </div>
     );
   }

   export default RecipeCard;
   ```

2. **Verwende die Komponente in `App.jsx`**: Wir importieren die `RecipeCard`-Komponente und können sie dann wie einen HTML-Tag verwenden.

   ```
   
   ```

```jsx
// src/App.jsx
import RecipeCard from './components/RecipeCard'; // Importieren

function App() {
  return (
    <div>
      <h1>Meine Lieblingsrezepte</h1>
      <RecipeCard />  {/* Wiederverwendbar */}
      <RecipeCard />
      <RecipeCard />
    </div>
  );
}

export default App;
```

Das Ergebnis im Browser sind die Überschrift und drei identische Rezeptkarten untereinander.

6. Kernkonzept 3: Props - Daten an Komponenten übergeben

Unsere `RecipeCard`-Komponente ist noch statisch. Damit sie dynamisch wird, müssen wir ihr Daten von außen übergeben. Das geschieht mit **Props** (kurz für Properties). Props werden von einer Elternkomponente an eine Kindkomponente "heruntergereicht".

- **Props sind schreibgeschützt (read-only)**. Eine Komponente darf ihre eigenen Props niemals verändern.

Beispiel: `RecipeCard` dynamisch machen

1. **Props in `App.jsx` übergeben**: Wir übergeben Daten an `RecipeCard` wie HTML-Attribute.

```jsx
// src/App.jsx
import RecipeCard from './components/RecipeCard';

function App() {
  return (
    <div>
      <h1>Meine Lieblingsrezepte</h1>
      <RecipeCard
        title="Spaghetti Carbonara"
        description="Ein klassisches italienisches Gericht."
        duration={30}
      />
      <RecipeCard
        title="Kaiserschmarrn"
        description="Eine süße österreichische Spezialität."
        duration={25}
      />
    </div>
  );
}

export default App;
```

2. **Props in `RecipeCard.jsx` empfangen und verwenden**: Die Props kommen als erstes Argument (ein Objekt) in die Komponentenfunktion.

```jsx
// src/components/RecipeCard.jsx

// 'props' ist ein Objekt: { title: "...", description: "...", duration: ... }
function RecipeCard(props) {
  return (
    <div className="recipe-card">
      <h2>{props.title}</h2>
```

```
      <p>{props.description}</p>
      <p>Zubereitungszeit: {props.duration} Minuten.</p>
    </div>
  );
}

export default RecipeCard;
```

- **Modernere Schreibweise mit Destructuring**: Es ist üblich, das `props`-Objekt direkt in der Funktionssignatur zu "entpacken" (destrukturieren), um den Code lesbarer zu machen.

```
// src/components/RecipeCard.jsx

function RecipeCard({ title, description, duration }) {
  return (
    <div className="recipe-card">
      <h2>{title}</h2>
      <p>{description}</p>
      <p>Zubereitungszeit: {duration} Minuten.</p>
    </div>
  );
}

export default RecipeCard;
```

  Das Ergebnis ist identisch, aber der Code ist sauberer.

7. Kurzer Ausblick: State mit `useState`

Props fließen immer von oben nach unten. Was aber, wenn eine Komponente ihren *eigenen*, internen Zustand verwalten muss, der sich durch Benutzerinteraktion ändert (z.B. der Text in einem Suchfeld oder ein Zähler)? Dafür gibt es in React **State**. Der `useState`-Hook ist das Werkzeug dafür.

Beispiel für einen einfachen Klick-Zähler:

```
import { useState } from 'react'; // Wichtig: Importieren!

function Counter() {
    // Erstellt eine Zustandsvariable 'count' und eine Funktion 'setCount', um sie zu ändern.
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>Du hast {count} Mal geklickt.</p>
            <button onClick={() => setCount(count + 1)}>
                Klick mich
            </button>
        </div>
    );
}
```

Dieses Konzept werden wir im nächsten Skript ausführlich behandeln.

Fazit

- **React ist deklarativ und komponenten-basiert**: Wir beschreiben, *was* die UI zeigen soll, indem wir sie in wiederverwendbare Bausteine zerlegen.
- **`Vite` ist unser Werkzeug**: Es erstellt die Projektstruktur und gibt uns einen schnellen Entwicklungsserver.
- **JSX ist HTML mit Superkräften**: Es erlaubt uns, die UI-Struktur direkt in JavaScript zu schreiben und mit JavaScript-Logik (`{}`) zu verbinden.

- **Komponenten sind Funktionen**: Eine JavaScript-Funktion, deren Name großgeschrieben wird und die JSX zurückgibt.
- **Props sind der Datenfluss**: Sie erlauben es uns, Daten von Eltern- zu Kindkomponenten zu übergeben und unsere UI dynamisch zu machen.

Cheat Sheet

- **Vite-Projekt erstellen**:

```
npm create vite@latest
```

- **Projekt starten**:

```
cd projekt-name
npm install
npm run dev
```

- **Named Export**: `export const myVar = ...;`
- **Named Import**: `import { myVar } from './myFile.js';`
- **Default Export**: `export default myFunction;`
- **Default Import**: `import MyFunction from './myFile.js';`
- **React-Komponente**: Eine Funktion mit großem Anfangsbuchstaben, die JSX zurückgibt.

```
function MyComponent({ prop1, prop2 }) {
  return <h1>Hallo, {prop1}!</h1>;
}
export default MyComponent;
```

- **JSX-Syntax**:
  - JavaScript in `{}`
  - HTML-Attribute: `className`, `htmlFor`
  - Ein Wurzelelement (`<div>` oder `<>`)

Übungsaufgaben

1. **Module erstellen und verwenden**:
   - Erstelle eine neue Datei `src/utils/text.js`.
   - Erstelle darin zwei **named exports**: eine Funktion `truncate(text, length)` die einen Text kürzt, und eine Funktion `shout(text)`, die einen Text in Großbuchstaben mit drei Ausrufezeichen zurückgibt.
   - Erstelle eine neue Datei `src/utils/greeting.js`. Erstelle darin einen **default export**: eine Funktion `getGreeting(name)`, die `"Hallo, [name]!"` zurückgibt.
   - Importiere und verwende alle drei Funktionen in deiner `App.jsx`, um die Ergebnisse in der Konsole (`console.log`) auszugeben.
2. **Eigene statische Komponente erstellen**:
   - Erstelle eine neue Komponente `src/components/Footer.jsx`.
   - Diese Komponente soll ein `<footer>`-Element mit einem Copyright-Hinweis (z.B. `© 2025 Recipe Platform`) und deinem Namen zurückgeben.
   - Importiere und verwende diese `Footer`-Komponente am Ende deiner `App.jsx`.
3. **Komponente mit Props dynamisch machen**:
   - Erstelle eine Komponente `src/components/UserProfile.jsx`.
   - Sie soll die Props `username` (String), `email` (String) und `isLoggedIn` (Boolean) akzeptieren.
   - In der Komponente soll der `username` und die `email` angezeigt werden.

- Zusätzlich soll mit einer bedingten Anweisung (`if` oder ternärer Operator) geprüft werden: Wenn `isLoggedIn` `true` ist, soll "Status: Online" in grün angezeigt werden, ansonsten "Status: Offline" in rot.
- Verwende diese Komponente mehrmals in `App.jsx` mit unterschiedlichen Werten für die Props.

4. **Komponenten verschachteln (Komposition)**:
   - Erstelle eine kleine Komponente `src/components/Avatar.jsx`. Sie soll eine Prop `imageUrl` erhalten und ein `<img>`-Element zurückgeben, dessen `src`-Attribut auf die `imageUrl` gesetzt ist.
   - Passe die `UserProfile`-Komponente aus der vorigen Aufgabe an: Sie soll nun zusätzlich die `Avatar`-Komponente enthalten und die `imageUrl`-Prop an sie weitergeben.
   - Passe den Aufruf in `App.jsx` an, um eine `imageUrl`-Prop (z.B. von einer Platzhalter-Bild-Webseite wie `https://via.placeholder.com/150`) an `UserProfile` zu übergeben.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Wir legen das Fundament für das Frontend unserer Rezept-Plattform.

Aufgabe:

1. **React-Projekt einrichten**: Erstelle ein neues Vite-React-Projekt namens `recipe-frontend`. Installiere die Abhängigkeiten und starte den Entwicklungsserver.
2. **Projekt aufräumen**: Lösche den Inhalt der `App.css` und entferne das meiste aus der `App.jsx`, sodass nur eine leere Grundstruktur übrig bleibt:

```jsx
// src/App.jsx
function App() {
  return (
    <div>
      <h1>Recipe Sharing Platform</h1>
    </div>
  )
}
export default App
```

3. **Statische Komponenten erstellen**:
   - Erstelle eine `Header.jsx`-Komponente, die eine Navigationsleiste mit Links (vorerst mit `#` als `href`) für "Home", "Alle Rezepte" und "Login" anzeigt.
   - Erstelle eine `Footer.jsx`-Komponente mit einem Copyright-Hinweis.
   - Importiere und verwende beide Komponenten in `App.jsx`, sodass die Seite einen Header, die `<h1>`-Überschrift und einen Footer hat.
4. **`RecipeListItem`-Komponente erstellen**:
   - Erstelle eine neue Komponente `RecipeListItem.jsx`.
   - Diese soll die Props `title` und `author` akzeptieren.
   - Sie soll ein `<li>`-Element zurückgeben, das den Titel und den Autor anzeigt, z.B. `"Spaghetti Bolognese (von Max Mustermann)"`.
5. **Rezeptliste in `App.jsx` erstellen**:
   - Füge in `App.jsx` eine `<h2>`-Überschrift "Beliebte Rezepte" hinzu.
   - Erstelle eine `<ul>`-Liste.
   - Verwende deine `RecipeListItem`-Komponente mehrmals **statisch** innerhalb der `<ul>`-Liste mit unterschiedlichen hartcodierten `title`- und `author`-Props, um eine Liste von Rezepten zu simulieren.
   - Das Ergebnis sollte eine Seite sein, die einen Header, eine Überschrift, eine Liste mit mehreren Rezepten und einen Footer anzeigt.