

RICH CLIENT REACT

LERNZIELE

- Wie schreiben wir eine SPA in Javascript?
- Wie unterstützt uns React beim Schreiben einer SPA?
- Wie hilft eine Component Architecture beim Schreiben eines Frontends?
- Wie bringe ich Ordnung in eine Component Architectur?
- Was sind Micro Frontends und was bringt das?

SPA IN PLAIN JAVASCRIPT

Speaker notes

- Bevor wir jetzt direkt in React einsteigen, lasst uns erstmal herausfinden, wofür React überhaupt gut ist.

SPA IN PLAIN JAVASCRIPT?

- Spricht etwas dagegen?
- Wäre quasi nur das exzessive Einsetzen von Ajax

Speaker notes

- Hat jemand Bedenken, wenn wir das machen?

PRAXIS: SPA IN PLAIN JAVASCRIPT

- Todo App als SPA in plain JavaScript
- HTML Seite mit plain Javascript anlegen
- Den Web Service habt ihr letzte Woche bereits geschrieben
- Falls ihr nicht fertig geworden seid
 - https://gitlab.com/dhbw_webengineering_2/rich_client_server

Speaker notes

- Also Web Service starten, HTML Seite anlegen und loslegen.

PRAXIS: ANFORDERUNGEN

- Listenansicht der Todos
 - Alle Titel der Todos werden in einer Liste angezeigt
 - Es gibt einen Button hinter jedem Todo, um eine Detailansicht zu öffnen
- Detailseite für ein Todo
 - Hier soll eine genauere Beschreibung des Todos angezeigt werden
- Kein explizites Routing (Anpassung der URL)

Speaker notes

- Wir schreiben nur zwei Seiten und das auch gerne ohne viel Styling.
- Es geht mir nur darum, mal zu zeigen wie man sowas ohne ein Framework bauen würde.
- Beispielanwendung zeigen, wie dies aussehen könnte.

PRAXIS: WAS BRAUCHEN WIR?

Speaker notes

- In der index.html Datei brauchen wir einen Bereich, in dem unsere zwei Seiten angezeigt werden.
- Möglich wäre auch ein Menü, dass auf beiden Seiten existiert, um auf die Startseite zurück zu kehren.
- XMLHttpRequests kennt ihr bereits aus der letzten Vorlesung, die könnt ihr hier anwenden.
- Per Javascript müssen wir dann das HTML so editieren, dass die Daten vom Backend entsprechend angezeigt werden können.
- Wer die Syntax zum editieren von HTML nicht mehr kennt, kann sich gerne des Internets bedienen.
- Die Listenansicht soll initial angezeigt werden, das heißt wir brauchen eine Art Initialisierung.

PRAXIS: WAS BRAUCHEN WIR?

- index.html mit Placeholder für den Inhalt

PRAXIS: WAS BRAUCHEN WIR?

- index.html mit Placeholder für den Inhalt
- HTTP call ans Backend, zum laden der Daten

PRAXIS: WAS BRAUCHEN WIR?

- index.html mit Placeholder für den Inhalt
- HTTP call ans Backend, zum laden der Daten
- Javascript zum Bauen der Listenansicht

PRAXIS: WAS BRAUCHEN WIR?

- index.html mit Placeholder für den Inhalt
- HTTP call ans Backend, zum laden der Daten
- Javascript zum Bauen der Listenansicht
- Javascript zum Bauen der Detailansicht

PRAXIS: WAS BRAUCHEN WIR?

- index.html mit Placeholder für den Inhalt
- HTTP call ans Backend, zum laden der Daten
- Javascript zum Bauen der Listenansicht
- Javascript zum Bauen der Detailansicht
- Initialisierung der Seite

PRAXIS: UNTERSTÜTZUNG

- Ihr könnt natürlich direkt mit einer HTML Seite starten
- Wer sich beim Styling ein wenig Zeit sparen möchte:
 - https://gitlab.com/dhbw_webengineering_2/spa_plain_javascript (branch: main)
 - Enthält Funktionen, um einige Components zu bauen

PRAXIS: LÖSUNG

- Lösung gibt es hier:
https://gitlab.com/dhbw_webengineering_2/spa_plain_javascript
(branch: solution)
- Ihr dürft euch natürlich inspirieren lassen

Speaker notes

- Meine Beispiellösung ist nicht perfekt.
- Versucht gerne eine bessere oder schönere Lösung zu bauen!

SPA IN PLAIN JAVASCRIPT?

Was ist bei einer Implementierung aufgefallen in plain JavaScript?

Speaker notes

- Das schreiben einer SPA in plain JavaScript ist sehr aufwendig.
- Backend Requests, DOM Manipulation sind Aufgaben, die in jeder Application auftreten. Braucht man nicht immer neu zu schreiben.
- Routing ist ebenfalls aufwendig. Umbau der Seite, Anpassung der URL.

SPA IN PLAIN JAVASCRIPT?

Was ist bei einer Implementierung aufgefallen in plain JavaScript?

- Es ist sehr aufwendig
- Viel boilerplate Code
 - XMLHttpRequests sind immer gleich
 - DOM Manipulation
 - Routing

SPA IN REACT

WIE HILFT UNS REACT?

- Unterstützt
 - DOM Manipulation
 - Routing
- Bietet eine Menge Libraries zur Unterstützung
 - Axios: für XMLHttpRequests
- Bietet Change Detection

Speaker notes

- React macht vieles einfacher, was wir in plain JavaScript selbst erledigen mussten.
- DOM Manipulation wird durch die JSX Syntax wesentlich einfacher.
- Routing mit entsprechender URL Manipulation wird zum Kinderspiel.
- Axios ist eine vorgefertigtes Tooling, um XMLHttpRequests abzusenden.
- Change Detection ist schon mal ein Schlagwort, das ihr euch merken könnt. Auch darauf werde ich später noch genauer eingehen, wenn es wichtiger für uns wird.

REACT FUNCTIONS

- Enthalten Information und Logik zum Rendern der UI
- Eine Mischung aus Javascript und HTML (JSX)

```
1 export default function ReactFunction() {  
2     const name = 'World';  
3  
4     return <div>Hello {name}!</div>  
5 }
```

Speaker notes

- React funktions geben als return Wert eine Art HTML Syntax zurück.
- Es darf maximal ein toplevel Tag geben.
- Über geschweifte Klammern können wir dynamische Daten in das HTML einfügen.
- Das macht das Rendering von dynamischen Informationen wesentlich einfacher als mit plain Javascript.

REACT FUNCTIONS

- Expressions im HTML sind möglich

```
1 export default function ReactFunction() {  
2     const names = ['World', 'Daniel', 'Iven', 'Kai'];  
3  
4     return <div>Hello {names.join(', ')}!</div>  
5 }
```

Speaker notes

- In den geschweiften Klammern können wir nicht nur Variablen einbinden, sondern auch verschiedene Expressions nutzen.

REACT FUNCTIONS

- HTML in einer Expression ebenfalls

```
1 export default function ReactFunction() {  
2     const names = ['World', 'Daniel', 'Iven', 'Kai'];  
3  
4     return <div>Hello {names.map(name => <b>{name}</b>)}!</div>  
5 }
```

Speaker notes

- Auch in den geschweiften Klammern können wir über die JSX Syntax wieder HTML Tags einfügen.

REACT FUNCTIONS

```
1 export default function ReactFunction() {  
2     let name;  
3  
4     return <div>{ name ? `Hello ${name}!` : 'Loading' }</div>  
5 }
```

PRAXIS: REACT FUNCTIONS (LISTVIEW)

- ListView mit React bauen
- Ganz simpel, kein Styling, kein Backend!
- https://gitlab.com/dhbw_webengineering_2/rich_client_react
(branch: step_0-list_view)

REACT ROUTING

- Bietet uns einfache Navigation
- Automatische Anpassung der URL

REACT ROUTING

```
1 function App() {
2   return (
3     <div className="App">
4       <BrowserRouter>
5         <Routes>
6           <Route path="/" element={<Screen1 />} />
7           <Route path="/screen1" element={<Screen1 />} />
8           <Route path="/screen2/:someParam"
9             element={<Screen2 />} />
10        </Routes>
11      </BrowserRouter>
12    </div>
13  );
14 }
```

Speaker notes

- Mit dem BrowserRouter wird eine Stelle markiert, an der die einzelnen Seiten angezeigt werden.
- Die Routes definieren alle Seiten, zu denen navigiert werden kann.
- Mehrere Pfade können zu gleichen Seite führen.
- Der Parameter "element" einer Route, definiert die React function, zu der navigiert wird.
- Über ein :someParam kann ein Parameter bei der Navigation mitgegeben werden.

REACT ROUTING

- Mit `useNavigate()` können wir Navigationen auslösen.

```
1 export default function SomeScreen() {
2   const navigate = useNavigate();
3   const someParam = 'someParam';
4
5   return <div>
6     <Button label='Screen1' onClick={navigate('/') }
7     <Button
8       label='Screen2 '
9       onClick={navigate(`/screen2/${someParam}`) }
10   </div>
11 }
```

Speaker notes

- Über die Funktion "useNavigate()" bekommt man eine Funktion, mit der man eine Navigation auslösen kann.
- Dieser Funktion übergibt man nun den Pfad. Dabei können natürlich auch Parameter eingefügt werden.

REACT ROUTING

- useParams() erlaubt es uns auf Pfadparameter zuzugreifen

```
1 export default function Screen2() {  
2     const { someParam } = useParams();  
3  
4     return <div>{someParam}</div>;  
5 }
```


Speaker notes

- Über die Funktion "useParams()" bekommt man ein Objekt mit allen übergebenen Parametern.
- Der Name des Parameters ist dabei der, der in der Route hinter dem Doppelpunkt definiert wurde.
- Über Javascript Destructuring wird der Parameter hier aus dem Objekt destructed.

PRAXIS: REACT ROUTING

- Zu unserem bestehenden ListView bauen wir einen DetailView
- Navigation zum DetailView und zurück soll möglich sein
- Wer nicht mitgekommen ist:
 - https://gitlab.com/dhbw_webengineering_2/rich_client_react (branch: step_1-routing)

LADEN DYNAMISCHER DATEN

- Mit der Library axios
- GET Request wie folgt:

```
1 async getData() {  
2     return axios.get(  
3         'https://somedomain.de/get/data',  
4     ).then((response) => response.data);  
5 }
```

Speaker notes

- Wie ihr sehen könnt, brauchen wir nur die URL.
- Als Response bekommen wir ein Javascript Objekt, welches axios bereits aus dem JSON geparsed hat.

LADEN DYNAMISCHER DATEN

- POST Request:

```
1 async saveData(data) {  
2     return axios.post(  
3         'https://somedomain.de/post/data',  
4         data,  
5         {  
6             headers: { 'Content-Type': 'application/json' },  
7         },  
8     ).then((response) => response.data);  
9 }
```

Speaker notes

- Auch POST Requests funktionieren mit axios ziemlich einfach.
- Wir können das Javascript Objekt, das gespeichert werden soll, einfacher als solches übergeben.
- Axios kümmert sich um die JSON Serialisierung.

LADEN DYNAMISCHER DATEN

- Auslagern der Requests in eine eigene Klasse

```
1 export default class DataHttpClient {  
2     async getData() {  
3     }  
4  
5     async saveData(data) {  
6     }  
7 }
```

Speaker notes

- Um die Funktionen zum Laden und Speichern von Daten auf verschiedenen Seiten nutzen zu können, sollten wir sie in eine Klasse auslagern.
- Außerdem bleiben so die React functions frei von solcher Logik und befassen sich nur mit dem Rendering

LADEN DYNAMISCHER DATEN

- Bereitstellen des DataHttpClient mittels Dependency Injection
- In React nutzt man Context Injection

```
1 export const DataHttpClientContext =  
2     createContext(DataHttpClient);  
3  
4 function App() {  
5     return (  
6         <div classname="App">  
7             <DataHttpClientContext.Provider  
8                 value={new DataHttpClient()}>  
9                 <Screen1 />  
10            </DataHttpClientContext.Provider>  
11        </div>  
12    );  
13 }
```

Speaker notes

- Braucht ihr noch mal eine Erläuterung zu Dependency Injection?
- Über den React Context können wir gewisse Objekte in Teilen des DOM's verfügbar machen.
- Mit der Funktion "createContext" erstellt man ein Kontext-Objekt von einem gewissen Klassentyp.
- Nun hängen wir einen Provider dieses Kontexts in dem DOM ein.
- Dieser Provider erhält nun das konkrete Objekt, das unter dem Provider im DOM verfügbar ist.

LADEN DYNAMISCHER DATEN

- Abrufen des Objekts mit useContext()
- Nur möglich, wenn sich die React function im korrekten Kontext befindet

```
1 export default function Screen1() {  
2     const dataHttpClient = useContext(DataHttpClientContext)  
3  
4     return <div></div>;  
5 }
```

PRAXIS: LADEN DYNAMISCHER DATEN

- Schreibt euch einen TodoHttpClient, mit dem ihr Todos abrufen könnt
- Startet dazu den Web Service vom letzten Mal
- Macht den Client per DI verfügbar
- Wer nicht mitgekommen ist:
 - https://gitlab.com/dhbw_webengineering_2/rich_client_react (branch: step_2-load_data)

REACT HOOKS

- Ein Thema für sich
- Speichern von State: `useState()`
- Lifecycle: `useEffect()`

Speaker notes

- React Hooks sind schon fast ein Thema für sich.
- Wenn wir Daten in einer React function speichern wollen, können wir das mit `useState()` -> gleich mehr dazu
- In unserer Plain Javascript SPA haben wir auch Code für die Initialisierung gebraucht. Bzw. wir haben die einzelnen Seiten initial geladen.
- In einer React function können wir nicht einfach asynchronen Code ausführen. Es muss immer direkt etwas zum Rendern zurückgegeben werden.
- Daher gibt es Effects, um asynchron etwas vom Backend zu laden. Hier greifen wir dann auf den Client den wir geschrieben haben zu.

REACT HOOKS

- `useState()`
 - Zum Speichern/Ändern von Daten in einer React function

```
1 export default function Screen2() {  
2   const [count, setCount] = useState(0)  
3  
4   return <button onClick={() => setCount(count + 1)}>  
5     {count}  
6   </button>;  
7 }
```

Speaker notes

- Mit der Funktion `useState()` können wir Daten in einer React function speichern.
- Die `useState()` function gibt uns ein Attribut "count" über das wir auf den Wert zugreifen können.
- Der zweite Rückgabewert ist eine function, mit der wir den Wert verändern können.
- Benennen können wir die Rückgabewerte, wie wir wollen. (hier wird wieder mit destruction gearbeitet)
- Eine normale Variable würde ihren Wert verlieren, wenn die React function erneut aufgerufen wird.
- `useState()` erzeugt einen state für die function der besteht. Außerdem wird die React function neu gezeichnet, sobald sich ein state ändert.
- Auch über das `onChange` event des input fields wird der state neu gesetzt.
- Angezeigt wird der aktuelle Wert, in dem wir die "count" variable als Label des Buttons anzeigen.

REACT HOOKS

- `useEffect()`
 - Seiteneffekte für React functions
 - Callback der zu bestimmten Zeitpunkten aufgerufen wird

```
1 export default function Screen2() {  
2     ...  
3  
4     useEffect(() => {  
5         dataHttpClient.getData()  
6             .then((data) => console.log(data));  
7     });  
8  
9     ...  
10 }
```

Speaker notes

- Mit der Funktion `useEffect()` definieren wir einen Callback, der zu gewissen Zeitpunkten automatisch von React angerufen wird.
- In diesem Beispiel wird der Callback zum Zeitpunkt der Initialisierung der React function aufgerufen.
- Die React function wird ausgeführt und anschließend wird der Request gegen das Backend gestartet.

REACT HOOKS

- `useEffect()`
 - Funktioniert gut in Kombination mit `useState()`

```
1 export default function Screen2() {  
2   ...  
3   const [data, setData] = useState('')  
4  
5   useEffect(() => {  
6     dataHttpClient.getData(data)  
7       .then((data) => setData(data));  
8   });  
9  
10  return <input  
11    value={data}  
12    onChange={(e) => setData(e.target.value)} />;  
13 }
```

Speaker notes

- `useEffect()` wird nicht nur zur Initialisierung der React function aufgerufen, es wird auch aufgerufen, wenn sich ein state ändert.
- Dieser Code würde daher eine Endlosschleife auslösen. Jedes mal wenn "setData" aufgerufen wird, wird der Callback erneut ausgeführt.

REACT HOOKS

- `useEffect()`
 - Muss nicht auf State Änderungen reagieren
 - Reagiert auf alle Parameter im Array

```
1 export default function Screen2() {
2   ...
3   const [data, setData] = useState('')
4
5   useEffect(() => {
6     dataHttpClient.getData(data)
7       .then((data) => setData(data));
8   }, []);
9
10  return <input
11    value={data}
12    onChange={(e) => setData(e.target.value)} />;
13 }
```

Speaker notes

- `useEffect()` löst nun keine Endlosschleife mehr aus.

REACT HOOKS

- `useEffect()`
 - Kann zum Aufräumen verwendet

```
1 export default function Screen2() {
2   ...
3   const [data, setData] = useState('')
4
5   useEffect(() => {
6     dataHttpClient.getData(data)
7       .then((data) => setData(data));
8     return () => console.log('teardown');
9   }, []);
10
11   return <input
12     value={data}
13     onChange={(e) => setData(e.target.value)} />;
14 }
```

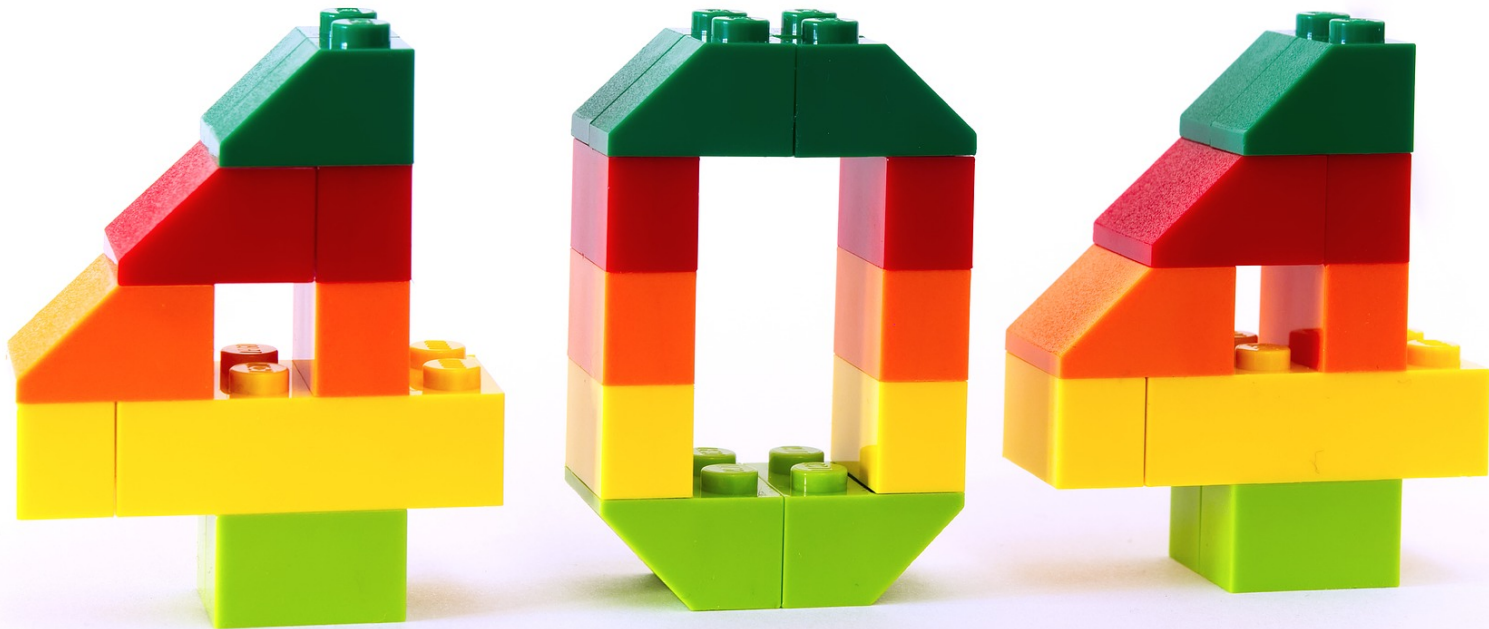
Speaker notes

- Der Rückgabewert des Callbacks des `useEffect()` Hooks wird aufgerufen, wenn die React function abgeräumt wird.

PRAXIS: REACT HOOKS

- Ladet die Todos über den Client mit dem useEffect() Hook
- Für den ListView und den DetailView
- Macht die Checkbox im Detailview funktionsfähig
- Wer nicht mitgekommen ist:
 - https://gitlab.com/dhbw_webengineering_2/rich_client_react (branch: step_3-react_hooks)

COMPONENT ARCHITECTURE



Speaker notes

- Component Architecture könnte man sich wie Lego vorstellen

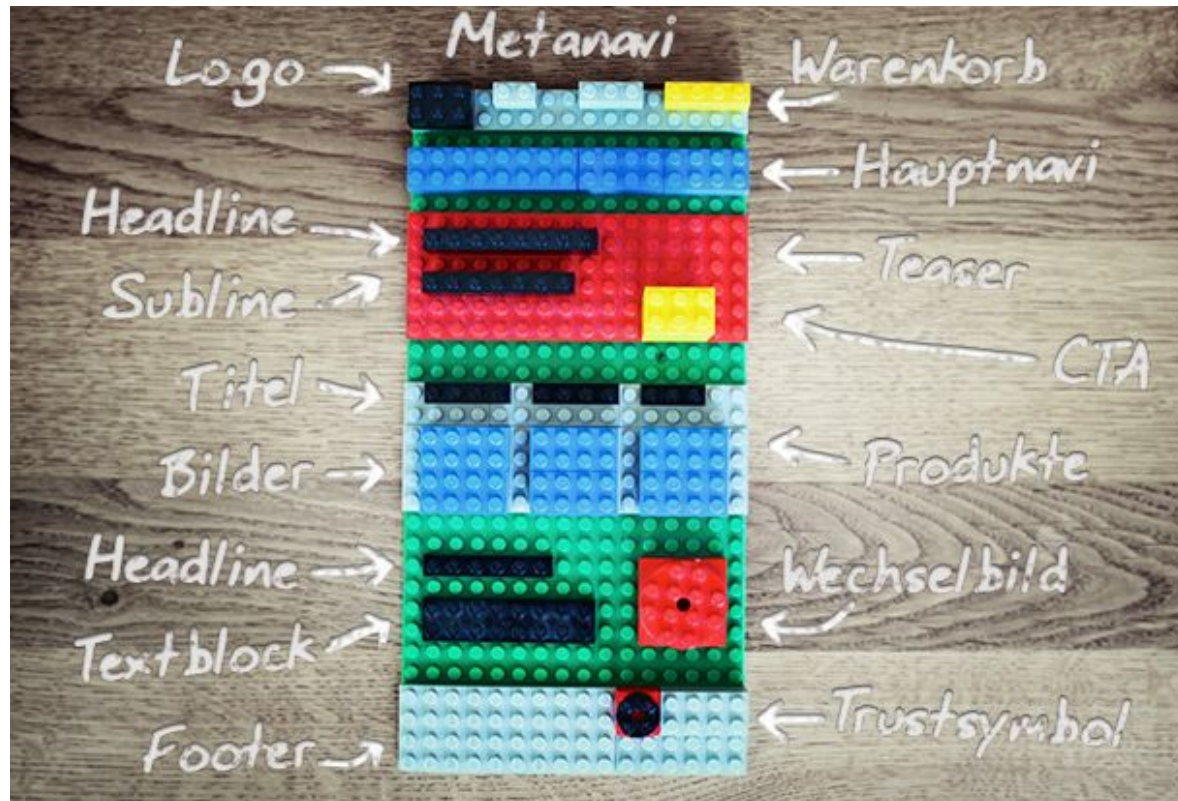
COMPONENT ARCHITECTURE

- Divide et impera
 - Teilen der Webseite in einzelnen Components
 - Verteilung und Strukturierung der Komplexität
- Components
 - Enthalten zusammengehörige Funktionalität
 - Haben feste Schnittstellen
 - Möglichst lose Kopplung und hohe Kohäsion
 - Analog wie Legosteine
 - Abstrahieren Struktur und Styling

Speaker notes

- Die Idee ist grundlegend, seine Webseite in einzelne Components aufzuteilen.
- Damit teilt man die Komplexität seiner Seite in kleinere Teile (Components).
- Quasi wie man es aus dem klassischen Softwareengineering kennt. Dort werden auch Funktionalitäten, die zusammengehören, in Komponenten zusammengefasst.
- Ob das nun Klassen sind (OOP) oder Funktionen, ist egal.
- Über Schnittstellen (vergleich zu Lego: die Noppen), können Components dann wieder zusammengesteckt werden.

COMPONENT ARCHITECTURE



COMPONENT ARCHITECTURE

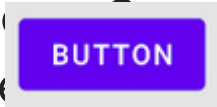
- Was kann alles eine Component sein?
 - Buttons, Text Fields, Labels, etc.
 - Search Bar, Form Groups, Cards, etc.
 - Header, Footer, Overlays, etc.
 - Pages

Speaker notes

- Unter einer Card kann man sich gebündelten Content vorstellen. Möglicherweise mit Bild und Edit Button oder so?
- Eine Component kann also ein sehr kleiner Teil der Anwendung sein, wie z.B. ein einzelner Button
- Eine Component kann aber auch ein Abschnitt sein oder sogar eine ganze Seite, die sich mit einem bestimmten Thema beschäftigt.

COMPONENT ARCHITECTURE

- Was kann alles eine Component sein?
 - Buttons, Text Fields, Labels, etc.
 - Search Bar, Form Groups, Cards, etc.
 - Header, Footer, Modals, etc.
 - Pages



COMPONENT ARCHITECTURE

- Was kenne ich aus dem Leben?
 - Bu
 - Se
 - He
 - Pa



Cafe Badilico

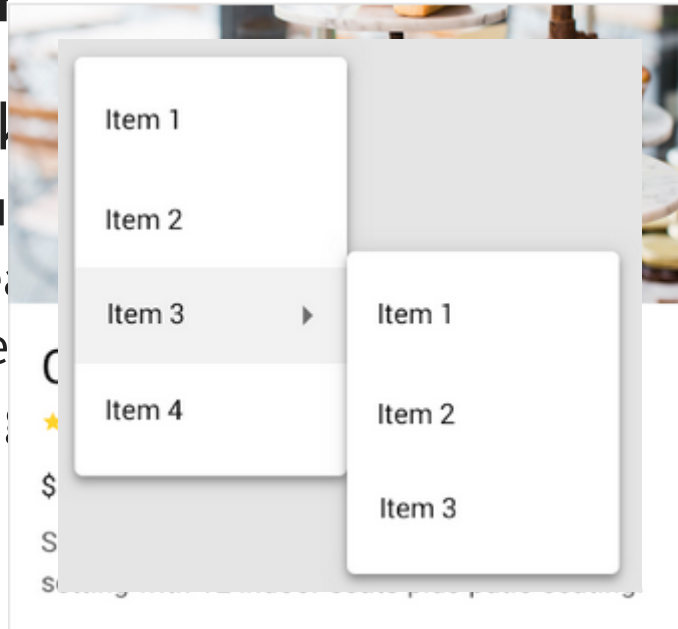
★★★★☆ 4.5 (413)

\$ • Italian, Cafe

Small plates, salads & sandwiches an intimate setting with 12 indoor seats plus patio seating.

COMPONENT ARCHITECTURE

- Was ist ein Component sein?
 - Button
 - Select
 - Header
 - Page



COMPONENT ARCHITECTURE

[Listenansicht](#)

Take the trash out

Done ☐

Details

Feed the cat

Done ☐

Details

Eat grandpa

Done ☒

Details

Clean the bathroom

Done ☐

Details

+

COMPONENTS

```
1 <button value="Submit" onclick="alert('Button clicked!')"/>
```

- Components haben feste Schnittstellen
- Damit können sie modular eingesetzt werden
- Es können Parameter rein und rausgegeben werden

Speaker notes

- Wir setzen mit der Component Architecture auf klassischen HTML Elementen auf und bauen daraus eigene Components

COMPONENTS IN REACT

- React functions sind Components
- Eine React function kann Parameter entgegennehmen
- Über einen Callback kann ein Wert zurückgegeben werden

```
1 export default function Button({ primary, label,  
2                                onClick, className }) => {  
3     const mode = primary ?  
4         'button--primary' : 'button--secondary';  
5     return (  
6         <button  
7             type="button"  
8             className={['button', mode, className].join(' ')  
9             onClick={() => onClick()}  
10        />  
11        {label}  
12    </button>  
13    );  
14 };
```

Speaker notes

- In diesem Beispiel geben wir mehrere Parameter in die Button function hinein.
- Primary ist ein bool, der Styling zum Button hinzufügt.
- Über das Label setzen wir einen Text.
- ClassName fügt weiteres Styling hinzu.
- OnClick ist ein Callback, mit dem wir zurückmelden können, wenn der Button geklickt wurde. Damit der Nutzer des Buttons weiß, was passiert.

COMPONENTS IN REACT

- Über propTypes können wir eine Schnittstelle definieren
- Über defaultProps können wir Defaultwerte hinterlegen

```
1 export default function Button({ ... }) {  
2     ...  
3 };  
4 Button.propTypes = {  
5     primary: PropTypes.bool,  
6     label: PropTypes.string.isRequired,  
7     onClick: PropTypes.func,  
8     className: PropTypes.string,  
9 };  
10 Button.defaultProps = {  
11     primary: false,  
12     onClick: undefined,  
13     className: '',  
14 };
```

Speaker notes

- Damit der Nutzer des Buttons weiß, was er hineingeben kann und muss, können wir die propTypes verwenden.
- Wir sollten eine klare Schnittstelle definieren, damit der Entwickler, der den Button nutzen möchte, nicht in die Implementierung schauen muss.

COMPONENT ARCHITECTURE

- Vorteile:
 - Konsistenz im Styling
 - Wiederverwendbarkeit
 - Schnellere Entwicklung
 - Einfachere Instandhaltung
- Nachteile:
 - Tiefe Verschachtelungen möglich

Speaker notes

- Konsistenz
 - Komponenten wie Buttons gehören zu Atomen und sollten wiederverwendet werden.
 - Dies spart Zeit, außerdem sehen die Button überall gleich aus. Sorgt für Konsistenz im Styling
- Schnellere Entwicklung
 - Ich muss den Button nicht noch mal für eine andere Seite stylen oder mit den Code dazu kopieren.
 - Ich kann auf bereits basierenden Strukturen aufbauen.
- tiefe Verschachtelungen
 - Große Seiten und Anwendungen kämpfen häufig mit einer sehr hohen Verschachtelungstiefe
 - Durch Komponenten, die kein Styling hinzufügen, sondern nur Logik bereitstellen und teilen, wird die Wrapper Hölle noch schlimmer.
 - Dies ist nicht sehr übersichtlich.

PRAXIS: COMPONENT ARCHITECTURE

- Aufteilen der ListView Seite in kleinere Components
- Überlegt euch selbst, wie ihr die Seite aufteilen könnt
- Basiskomponenten stehen bereit um Zeit zu sparen
 - https://gitlab.com/dhbw_webengineering_2/rich_client_react (branch: step_3-component_architecture)

ATOMIC DESIGN

ATOMIC DESIGN

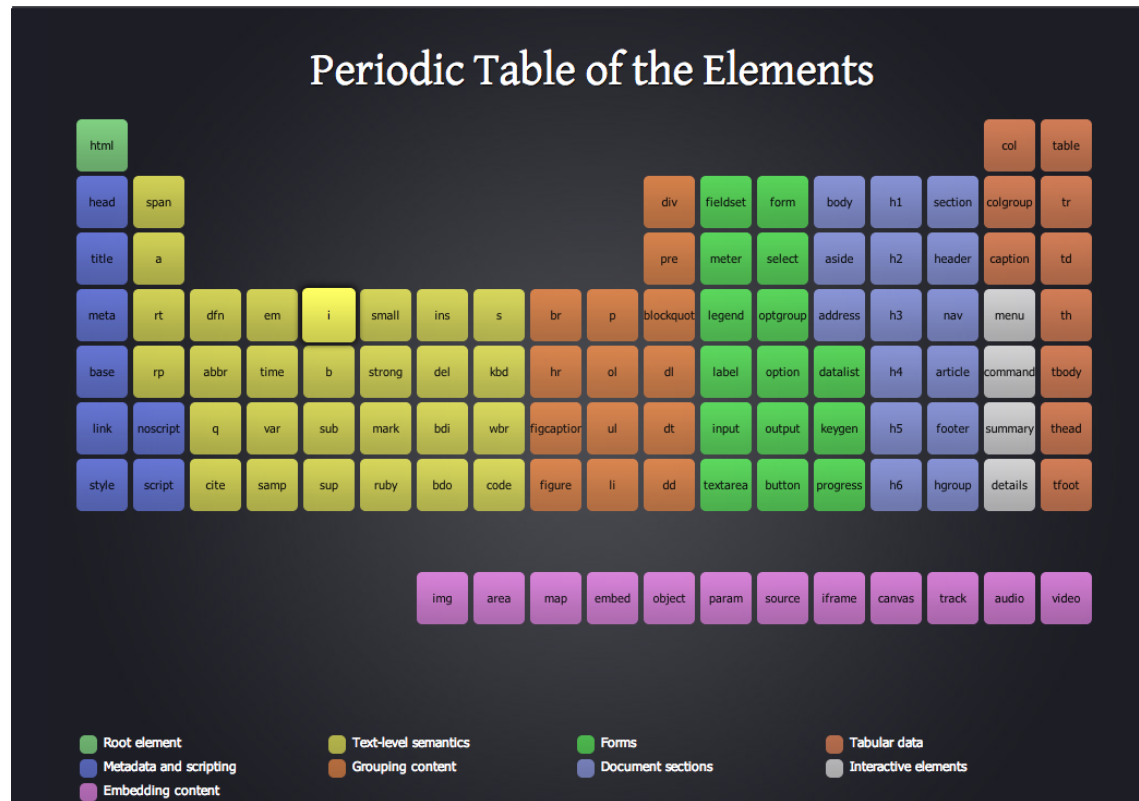
Große Frontends mit vielen Components werden unübersichtlich



ATOMIC DESIGN

Strukturierung und Kategorisierung von Components

Ziel ist ein ordentlicher Baukasten an Components



<https://bradfrost.com/blog/post/atomic-web-design/>

ATOMIC DESIGN

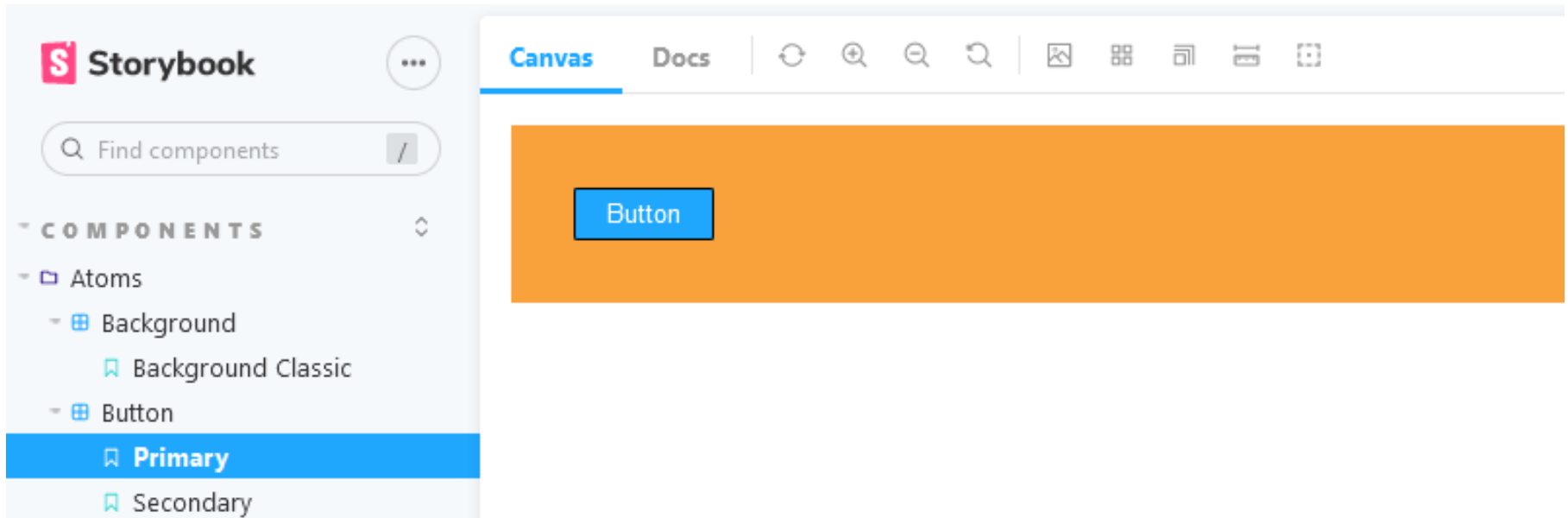
- Atomic Design ordnet Components nach:
 - Atoms - Buttons, Text Fields, etc.
 - Molecules - Search Bar, Form Groups, etc.
 - Organisms - Header, Footer, Overlays, etc.
 - Templates - Schablone
 - Pages - konkrete Seite

Speaker notes

- <https://bradfrost.com/blog/post/atomic-web-design/>
- Atoms - die Bausteine unserer Anwendung - Buttons, etc.
- Molecules - kleine Zusammenschlüsse von Atoms - Suchfelder, Form Groups
- Organisms - fachliche Components. Zusammenschlüsse von Molecules, mit denen der User interagieren kann
- Templates - Schablone die den Aufbau der Seite zeigt
- Pages - konkrete Seiten

STORYBOOK

- Macht Atomic Design noch nützlicher
- Visualisierung einzelner Components in verschiedener Ausprägung



STORYBOOK SYNTAX

```
1 export default {  
2   title: 'Components/Background',  
3   component: Background,  
4 };  
5  
6 const Template = () => <Background />;  
7  
8 export const BackgroundClassic = Template.bind({});
```

Speaker notes

- Ein Storybookeintrag besteht aus drei Bestandteilen.
- Definition des Eintrags
 - Hier geben wir dem Eintrag einen Titel. Dieser muss auch den Pfad enthalten, damit die Einträge entsprechend geordnet dargestellt werden.
 - Außerdem müssen wir hier die Component angeben, die wir in diesem Storybookeintrag anzeigen wollen. In diesem Fall die Background Component
- Definition des Templates
 - Im Template können wir mit der JSX Syntax definieren, was in der Story angezeigt wird.
 - Dies muss nicht immer nur die eine Component sein. Manchmal wollen wir evtl. noch einen Hintergrund hinzufügen, oder müssen etwas im Context mocken -> dazu gleich mehr
- Definition der konkreten Ausprägung
 - Es kann in einem Storybookeintrag für eine Component mehrere Ausprägungen einer Component geben.
 - Durch die Input Parameter der Component könnte die Component unterschiedlich aussehen.
 - BackgroundClassic ist in diesem Fall der Name der Ausprägung.
 - Mit `Template.bind({})` erstellen wir die Ausprägung. Gleich mehr dazu, wie man Parameter an die Component übergibt.

STORYBOOK SYNTAX

```
1 export default {  
2   title: 'Components/Background',  
3   component: Background,  
4 };  
5  
6 const Template = () => <Background />;  
7  
8 export const BackgroundClassic = Template.bind({});
```

STORYBOOK SYNTAX

```
1 export default {  
2   title: 'Components/Background',  
3   component: Background,  
4 };  
5  
6 const Template = () => <Background />;  
7  
8 export const BackgroundClassic = Template.bind({});
```

STORYBOOK SYNTAX

```
1 export default {  
2   title: 'Components/Background',  
3   component: Background,  
4 };  
5  
6 const Template = () => <Background />;  
7  
8 export const BackgroundClassic = Template.bind({});
```


STORYBOOK MIT PARAMETERN

```
1  const Template = (args) => <Background>
2                                <Button {...args} />
3                                </Background>;
4
5  export const Primary = Template.bind({});
6  Primary.args = {
7    primary: true,
8    label: 'Button',
9  };
10
11 export const Secondary = Template.bind({});
12 Secondary.args = {
13   label: 'Button',
14 };;
```

Speaker notes

- In dieser Story werden Parameter an die Component übergeben.
- Bei der Definition des Templates werden die Parameter über `destruction` an die Component übergeben.
- In diesem Fall wurde auch neben der Component die in der Story präsentiert werden soll, noch ein Hintergrund eingefügt.
- `Template.bind({})` wird nun zweimal ausgeführt, da es zwei verschiedene Ausprägungen des Buttons gibt.
- Anschließend werden auf der Ausprägung die `"args"` gesetzt.
- Es gibt eine `"primary"` und eine `"secondary"` Ausprägung des Buttons.

STORYBOOK MIT PARAMETERN

```
1  const Template = (args) => <Background>
2                                <Button {...args} />
3                                </Background>;
4
5  export const Primary = Template.bind({});
6  Primary.args = {
7    primary: true,
8    label: 'Button',
9  };
10
11 export const Secondary = Template.bind({});
12 Secondary.args = {
13   label: 'Button',
14 };
```

STORYBOOK MIT PARAMETERN

```
1  const Template = (args) => <Background>
2                                <Button {...args} />
3                                </Background>;
4
5  export const Primary = Template.bind({});
6  Primary.args = {
7    primary: true,
8    label: 'Button',
9  };
10
11 export const Secondary = Template.bind({});
12 Secondary.args = {
13   label: 'Button',
14 };;
```

STORYBOOK MIT PARAMETERN

```
1  const Template = (args) => <Background>
2                                <Button {...args} />
3                                </Background>;
4
5  export const Primary = Template.bind({});
6  Primary.args = {
7    primary: true,
8    label: 'Button',
9  };
10
11 export const Secondary = Template.bind({});
12 Secondary.args = {
13   label: 'Button',
14 };;
```

STORYBOOK MIT CONTEXTINJECTION

```
1  const someMockClient = {
2    getData(_) {
3      return Promise.resolve(new Data('lala'));
4    },
5    saveData(data) {
6      return Promise.resolve(data);
7    }
8  }
9
10 const Template = (args) =>
11     <SomeContext.Provider value={someMockClient}>
12       <SomeComponent {...args} />
13     </SomeContext.Provider>;
```

Speaker notes

- Wenn eine Component auf den Context zugreift, um einen HttpClient oder ähnliches zu verwenden, dann müssen wir diesen mocken.
- Dazu können wir einfach einen Provider mit einem mockValue bereit stellen.

STORYBOOK MIT ROUTER

```
1 export const reactRouterDecorator = (Story) => {
2   return (
3     <MemoryRouter>
4       <Routes>
5         <Route path="/" element={<Story />} />
6       </Routes>
7     </MemoryRouter>
8   )
9 }
```

```
1 export default {
2   title: 'Components/SomeComponent',
3   component: SomeComponent,
4   decorators: [reactRouterDecorator],
5 };
```


Speaker notes

- Wenn eine Component, die wir in einem Storybookeintrag anzeigen wollen, routing enthält, müssen wir einen Router um die Component legen. Sonst führt dies zu Fehlern.
- Da wir hier kein spezielles Objekt mocken wollen, können wir einen generalisierten Decorator bauen, den wir über viele Stories legen können.
- Der `reactRouterDecorator` bekommt die Story als Parameter hineingegeben. Anschließend wird der Router um die Component gelegt.
- Über den parameter "decorators" können wir den Decorator zu einem Storybookeintrag hinzufügen.

PRAXIS: ATOMIC DESIGN + STORYBOOK

- Sortieren des Projekts nach Atomic Design
- Storybookeintrag erstellen für wenigstens drei weitere Components
- Wer nicht mitgekommen ist:
 - https://gitlab.com/dhbw_webengineering_2/rich_client_react (branch: step_3-atomic_design)

MICRO FRONTENDS

COMPONENT ARCHITECTURE

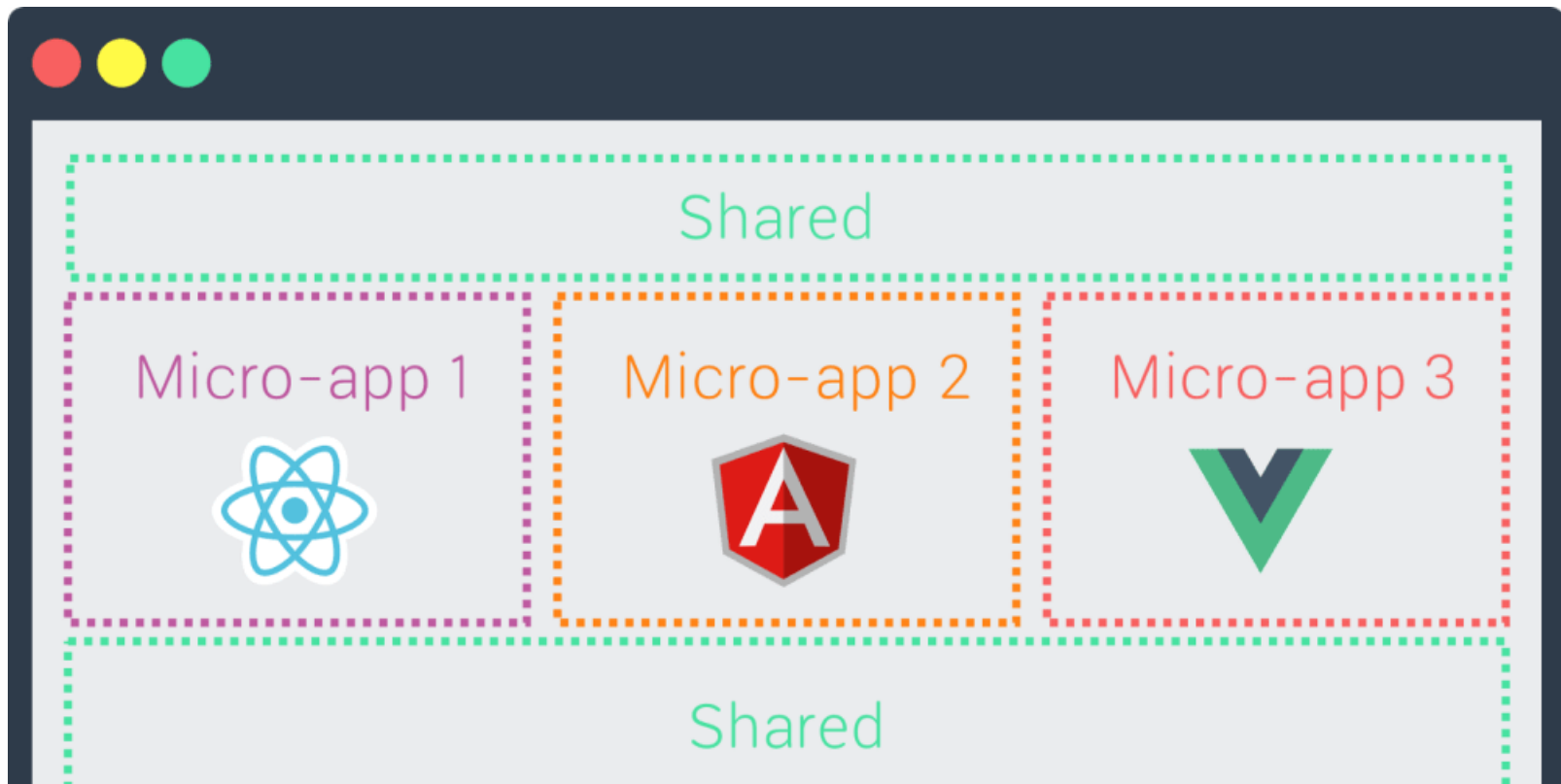
- Bringt uns Ordnung und Struktur
- Was passiert, wenn das Frontend wächst?
- Mehrere Teams arbeiten an einem Frontend?
- Unterschiedliche Teams
 - Mögen unterschiedliche Technologien
 - Haben unterschiedliche Arbeitsweisen
 - Möchten unabhängig releasen
 - Haben unterschiedlichen Codestyle

Speaker notes

- Auch bei einer ordentlich gepflegten Component Architecture wird ein Frontend irgendwann zu groß.
- Meistens wächst das Team ebenfalls, da die Anforderungen wachsen.
- Das Team teilt sich dann meist schon von selbst, da ein Einzelner nicht mehr die ganze Domäne überblicken kann.
- Irgendwann lohnt es sich das Team und das Frontend offiziell zu teilen.
- Sonst ist es absehbar, dass die Produktivität sinkt.
- Zum Beispiel durch:
 - Mergeconflicts durch zu viele Änderungen.
 - Konflikte in einem großen Team durch
 - Unterschiedlichen Codestyle.
 - Unterschiedliche Arbeitsweisen.
 - Unterschiedlich favorisierte Technologien.
 - Features können nicht unabhängig voneinander released werden.

MICRO FRONTENDS

- Aufteilen des Monolith in mehrere Frontends
- Frontends können zu einem Frontend zusammengesteckt werden

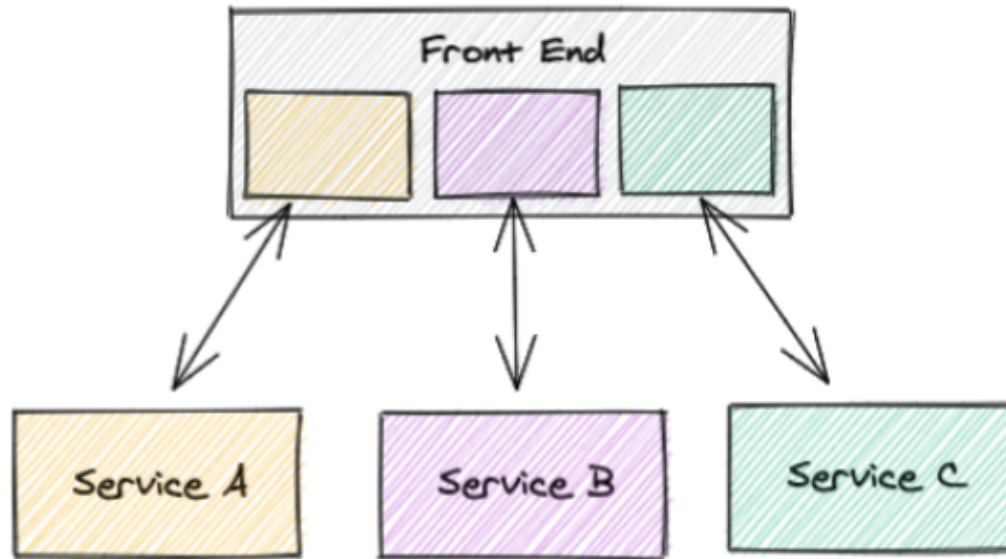


Speaker notes

- Micro Frontends können bei Bedarf zu einem Frontend zusammengesteckt werden.
- Dies muss aber nicht sein, evtl. wird über eine Navigation von einem zum anderen Frontend navigiert.
- Micro Frontends können von verschiedenen Teams mit verschiedenen Sprachen und Frameworks gebaut werden.
- Evtl. auch ein eigener release Zyklus.
- Micro Frontends können auch helfen, wenn man eine legacy Anwendung Stück für Stück erneuern möchte.

MICRO FRONTENDS

- Reden meist auch mit eigenen Backends
- Micro Services



Speaker notes

- Micro Frontends sind aus Micro Services entstanden
- Um auch die Frontends skalierbarer zu machen
- Dies ist nur bei wirklich komplexen Anwendungen zu empfehlen

LERNZIELE

- Wie schreiben wir eine SPA in Javascript?
- Wie unterstützt uns React beim Schreiben einer SPA?
- Wie hilft eine Component Architecture beim Schreiben eines Frontends?
- Wie bringe ich Ordnung in eine Component Architektur?
- Was sind Micro Frontends und was bringt das?

Speaker notes

- Eine SPA in plain Javascript zu schreiben funktioniert, ist allerdings umständlich.
- Da es Frameworks wie Sand am Meer gibt, können wir uns hier gut bedienen.
- React bietet uns viele verschiedene Tools, um uns das Leben leichter zu machen.
 - Model binding
 - Routing
 - Http calls
 - Components
- Mit einer Component Architecture schaffen wir uns einen Baukasten an Components, mit denen wir konsistent und schnell ein Frontend zusammenstecken können.
- Mit Atomic Design und Storybook bekomme ich Ordnung in meine Component Architecture, finde Components schneller, sehe sie alle auf einen Blick und beschleunige meine Arbeit.
- Wenn ein Frontend zu groß wird, das Team wächst oder man einen legacy Monolith Stück für Stück erneuern möchte, können Microfrontends nützlich sein.