1 Introduction SPA

1.1 Browser-based Applications

Benefits

- Work from anywhere, anytime
- Platform independent, including mobile
- No software update, no application, easy maintenance
- Software can be provided as a service (SaaS - pay as you go)
- Code separation

Liabilities

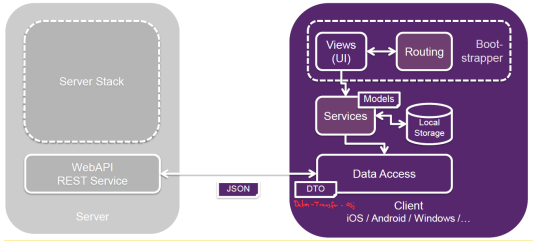
- No data sovereignty (Datenhoheit)
- Limited/restricted hardware access
- SEO - Search engines must execute JavaScript
- More complex deployment strategies

1.2 SPA

A website that fits on a single web page with a user experience similar to that of a desktop application. All code is retrieved with a single page load or resources are dynamically loaded. SPAs use AJAX and HTML5 to create responsive Web apps, without constant page reloads.

1.2.1 Architecture

Website interacts with user by rewriting parts of the DOM. After first load, all interaction with the server happens through AJAX.



1.2.2 Bundling

All JS code must be delivered to the client over potentially slow networks. Bundling and minifying the source leads to smaller SPA footprint. Larger SPAs with many modules need a reliable dependency management. Initial Footprint can be reduced by loading dependent modules on-demand.

1.2.3 WebPack as Bundler

Entry: Start, follows the graph of dependencies to know what to bundle.
Output: Tell webpack where to bundle your application.
Loaders: Transforms these files into modules as they are added to your dependency graph.
Plugins: Perform tasks like bundle optimization, asset management and injection of env variables.
Mode: Enable built-in optimization mechanisms.

1.3 Routing

- Completely on client-side by JS
- Navigation behaves as usual
- Browser needs to fake the URL to change and store page state
- *window.history.pushState*

1.4 Dependency Injection

Benefits

- Reduces coupling between consumer and implementation
- Contracts between classes are based on interfaces
- Supports the open/closed principle
- Allows flexible replacement of an implementation

1.4.1 Decorators

- Provide a way to add annotations / meta-programming syntax
- Can be attached to a class declaration, method, accessor, property or parameter
- Widely used in Angular

2 React

- Library, kein Framework
- Um User Interfaces zu bauen
- View in MVC
- Minimales Featureset
- Entwickelt von Facebook
- Verwendet für: WhatsApp, Insta, AirBnb, etc.

2.0.1 Prinzipien

- Komplexes Problem aufteilen in einfachere Komponenten
- Für eine bessere: Wiederverwendbarkeit, Erweiterbarkeit, Wartbarkeit, Testbarkeit, Aufgabenverteilung

2.1 Entwicklung von UIs

- Beschreibung des UIs
- Event-Handling
- Aktualisieren der Views

2.2 Komponenten und Elemente

- Funktionen die HTML zurückgeben
- Beliebige Komposition von React-Elementen und DOM-Elementen

```
function App() {
 return (
   <div>
     <HelloMessage name="HSR"/>
     <img src="/logo.png"/>
   </div>
 )
}
```

Parameterübergabe an Funktion

Äquivalent zu Attribut für DOM-Element

2.3 JavaScript XML

React verwendet JSX (blau), eine Erweiterung von JavaScript (gelb). Überall wo JSX verwendet wird, muss *react* importiert werden.

```
const menu = entries.map(entry =>
  <ListItem as="a" to={`/${entry.path}`}>
    <h1>{entry.title.toUpperCase()}</h1>
    <p>{entry.subtitle}</p>
  </ListItem>
)
```

Styles: werden nicht als Strings sondern als Object angegeben.

2.3.1 Conditionals

```
<Container>
 { error &&
   <Message>
     Fehler: {error}
   </Message>
 }
</Container>
```

oder

```
<Container>
 { error
   ? <span>
       Fehler: {error}
     </span>
   : <span>OK!</span>
 }
</Container>
```

2.3.2 Props

Komponenten erhalten alle Parameter/Properties als **props** Objekt.

- *this.props* bei Klassen
- Bei Funktionen als Parameter
- Immer **read-only**

2.3.3 Rendering und Mounting

Mounting: nötig um Komponenten auf Webseite anzuzeigen. *ReactDOM.render*

```
ReactDOM.render(
    <App/>
    document.getElementById('root')
)
```

2.4 React State

React-Klassenkomponenten können einen veränderbaren Zustand haben. Der **state** einer Komponente ist immer privat. Ändert der State, wird auch die Komponente aktualisiert.

```
class Counter extends React.Component {
    state = { counter: 0 }
    // ...
}
```

2.4.1 Event Handler

```
const increment = () => {
    this.setState({counter: this.state.counter +
        1})
} // ...
<button onClick={this.increment}>
```

2.5 Reconciliation

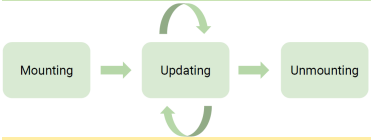
1. React Komponenten werden als virtueller DOM gerendert
2. Wird der **state** geändert, erstellt React einen virtuellen DOM
3. Alter und neuer DOM werden verglichen
4. Erst dann werden geänderte DOM-Knoten im Browser erstellt

2.6 Formulare

2.6.1 Input Handling

```
<form onSubmit={this.handleSubmit}>
<input value={this.state.username}
       onChange={this.handleUsernameChange} //...
</form>
handleUsernameChange = (event) => {
    this.setState({username: event.target.value});
};
handleSubmit = (event) => {
    event.preventDefault();
    //...
}
```

2.7 Komponenten Lifecycle



2.7.1 Mounting

1. **constructor(props)**
   - State initialisieren, sonst weglassen
2. static getDerivedStateFromProps(props, state)
   - Von State abhängige Props initialisieren
3. **render()**
4. **componentDidMount()**
   - DOM ist aufgebaut
   - Guter Punkt um zum Beispiel Async-Daten zu laden
   - setState Aufruf führt zu re-rendering

2.7.2 Updating

1. static getDerivedStateFromProps(props, state)
   - Von State abhängige Props aktualisieren
2. shouldComponentUpdate(nextProps, nextState)
   - wird false zurückgegeben wird render übersprungen
3. **render()**
4. getSnapshotBeforeUpdate(prevProps, prevState)
5. **componentDidUpdate(prevProps, prevState, snaspshot)**
   - Analog zu componentDidMount, DOM ist aktualisiert

2.7.3 Unmounting

1. **componentWillUnmount()**
   - Aufräumen

2.7.4 Error Handling

1. static getDerivedStateFromError(error)
   - Error im State abbilden
2. componentDidCatch(error, info)
   - Logging
   - Verhindern, dass Fehler propagiert wird, analog zu catch-Block in try-catch

2.8 React Router

- Komponentenbibliothek
- Komponenten anzeigen oder verstecken abhängig von der URL
- Für React Web und React Native

2.8.1 Router Komponenten

`<Router>`
Alle Routen müssen Teil des Routers sein, typischerweise nahe der Root-Komponente
`<Route exact path="/" component={Home} />`
Home-Komponente wird nur gerendert, wenn der path (exakt) matcht. Mehrere Route Elemente können gleichzeitig aktiv sein.
`<Link to="/">Home</Link>`
App-interne Links, welche nicht wie <a >die Seite neu laden.
`<Redirect to="/somewhere/else">`
Wird ausgeführt, sobald gerendert.

2.9 Hooks

Problem von Lifecycle Methoden Zusammengehörender Code ist auf mehrere Methoden verteilt (Mount/Unmount).
Problem von Klassen-State State ist über verschiedene Methoden verteilt
Fazit:

- Lifecycle und State ohne Klassen machen react verständlicher
- Klassen sind weiterhin unterstützt
- Hooks erlauben, Logik mit Zustand einfacher wiederzuverwenden

2.9.1 State Hook

```
function Counter() {
    const [count, setCount] = useState(0);
    // button => setCount(count + 1)
    return( <p>{count}</p> );
}
```

Mehrere State-Variablen: useState Aufrufe müssen immer in derselben Reihenfolge gemacht werden.

2.9.2 Effect Hook

```
useEffect(() => {
    // Mount stuff
    return () => {
        // Unmount stuff
    }
}, [] /* <= Dependencies */);
```

2.10 Flow

- Erweitert JavaScript um Typenannotationen
- Typ-Annotation im Code Typ-Inferenz für lokale Definitionen
- Generics, Maybe-Types, Union and Intersection-Types

2.11 TypeScript und React

- Mehr Typensicherheit in React-Komponenten
- Props und State lassen sich typisieren

Vorteil gegenüber Flow:

- Vollwertige Programmiersprache
- Besser unterstützt von Libraries und IDEs
- TypeScript Fehler müssen korrigiert werden

2.12 React Context

Ermöglicht es, Props für alle Unterkomponenten zur Verfügung zu stellen. (Theme Variablen)

```
// provider
const c = React.createContext(themes.light);
const theme = useContext(c); // consumer
```

2.13 Redux

Library für Statemanagement (Repräsentation / Veränderung / Benachrichtigung). State wird als Tree (immutable) von Objekten dargestellt. Veränderung am Tree führt durch den Reducer zu einem neuen Tree t+1 (funktionale Programmierung). State wird im **Store** verwaltet.

2.13.1 Actions

Benötigt um Stateänderungen zu machen. Wird an den Store gesendet / dispatched. Action ist eine reine Beschreibung der Action.

```
{type: 'TRANSFER', amount: 100 }
```

2.13.2 Reducer

Pure Funktionen, haben keine Seiteneffekte.

```
function balance(state = 0, action) {
    switch (action.type) {
        case 'TRANSFER':
            return (state + action.amount);
        default:
            return state;
    }
}
```

Reducer kombinieren: Jeder Reducer erhält einen Teil des States, für den er zuständig ist. Resultat wird in einem neuen State-Objekt kombiniert.

```
function rootReducer(state = {}, action) {
    return {
        balance: balance(state.balance, action),
        transactions: transactions(state.
            transactions, action)
    }
}
// Hilfsfunktion combineReducers:
const rootReducer = combineReducers({
    balance, transactions
});
```

2.13.3 Store erstellen

```
const store = createStore(rootReducer);
```

Mit dem root-Reducer kann der Store erstellt werden. In Kombination mit React führt das zu einem re-rendering der Komponenten.

2.14 React <3 Redux

Redux mit React verbinden:

```
const mapStateToProps = (state) => {
    return {
        transactions: state.transactions
    }
}
const mapDispatchToProps = {
    fetchTransactions
}
export default connect(mapStateToProps,
    mapDispatchToProps)(Component);

// Root Komponente
const store = createStore(
    rootReducer, applyMiddleware(thunkMiddleware));
render(
    <Provider store={store}>
        <App />
    </Provider>
    document.getElementById('root')
)
```

mapStateToProps: erhält State und kann daraus Props ableiten.
Die Komponente bekommt auch die *dispatch* Methode des Stores als Prop. Das Resultat von *connect* ist eine React-Komponente die mit dem Store verbunden ist.
Store muss der Root-Komponente mitgegeben werden.
thunkMiddleware: Erlaubt es, anstelle eines Objektes eine Funktion zu dispatchen (benötigt für asynchrone Actions).

2.14.1 Thunk Actions

```
function fetchTransactions(token) {
    return (dispatch, getState) => {
        dispatch({type: "FETCH_TRANSACTIONS_STARTED
            "});
        api.getTransactions(token)
            .then(({result: transactions}) => {
                dispatch({type: "
                     FETCH_TRANSACTIONS_SUCCEEDED",
                      transactions});
            })
    };
}
```

2.14.2 Selectors

Getter bei den Reducern, die einen Subtree des Stores zurückgeben. Wissen über den Aufbau des State-Trees bleibt bei den Reducern.