1 Introduction

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

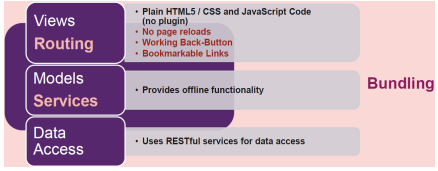
Liabilities:
- No data sovereignty (Datenhoheit)
- Limited/restricted hardware access
- SEO - Search engines must execute JavaScript
- More complex deployment strategies

1.2 SPA

In an SPA, either all necessary code is retrieved with a single page load or the appropriate resources are dynamically loaded and added to the page as necessary. Uses AJAX and HTML5

Traditional Architecture: Server renders a new HTML page with every call. (Major logic on server, no architectural separation between presentation and logic)

SPA Architecture: Website interacts with user by rewriting parts of the DOM (moves logic from server to client, server provides APIs like REST/GraphQL) → After first load, all interaction with the Server happens through AJAX



SPA aus Kundensicht: Sobald Desktop ähnliche User Experience gewünscht ist. Mehr möglichkeit für komplexe WebApps mit viel Animationen/graphischen Elementen.

Technischer Nutzen von SPA: Server App wird von Darstellung getrennt: Separation of Concerns, Bessere Wartbarkeit das Client-Codes, Aufteilung in Teams/Kompetenzzentren

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

Dependency Injection Benefits:
- Reduces coupling between consumer and the implementation
- The contracts between the classes are based on interfaces
- Classes relate to each other not directly, but mediated by thier interfaces
- Supports the open/closed principle
- Allows flexible replacement of an implementation

2 React

Ist eine Library von Facebook, um User Interfaces zu bauen.

Prinzipien: Komplexes Problem aufteilen in einfachere Komponenten. Bessere Wiederverwendbarkeit, Erweiterbarkeit, Wartbarkeit, Testbarkeit und Aufgabenverteilung

Komponenten und Elemente: Sind Funktionen, die HTML zurückgeben. Beliebige Komposition von React-elementen und DOM-Elementen:

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

2.1 JavaScript XML

React verwendet JSX (blau), eine Erweiterung von JavaScript (gelb). **Styles** werden nicht als Strings, sondern als Objekt mitgegeben

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

2.1.1 Conditionals

Was zu null, true, false oder undefined evaluiert wird nicht ausgegeben

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

oder

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

2.1.2 Funktions- und Klassenkomponenten

```
// Alle 3 äquivalent
function HelloMessage(props) {
  return <div>Hello {props.name}</div> }
class HelloMessage extends React.Component {
  render() {return <div>Hello {this.props.name}</div> }}
const HelloMessage = ({name}) => <div>Hello {name}</div>
```

2.1.3 Props

Komponenten erhalten alle Parameter/Properties als **props** Objekt. Bei Klasse als this.props, bei Funktionen als Parameter.
→ Props sind immer **read-only**

2.1.4 Rendering und Mounting

Mounting: Nötig um Komponenten auf Webseite anzuzeigen.

```
import React from 'react'
import ReactDOM from 'react-dom'
ReactDOM.render(
  <App/>, document.getElementById('root') )
```

2.2 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 }
  increment = () => { this.setState({
    counter: this.state.counter + 1 }) }
  render() { return (<div> {this.state.counter}
    <button onclick={this.increment}>Increment</button>
  </div>) } }
```

2.3 Formulare mit React

```
<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(); }
```

2.4 Komponenten Lifecycle

React Klassen-Komponenten haben Lifecycle-Methoden. Diese können in 3 Phasen unterteilt werden:

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 z.B. Async-Daten zu laden, setState Aufruf führt zu re-rendering

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, snapshot)** → Analog zu componentDidMount, DOM ist aktualisiert

Unmounting:
1. **componentWillUnmount()** → Aufräumen

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

2.5 React Router

Komponentenbibliothek, Komponenten anzeigen/verstecken abhängig von der URL, Für React Web und React Native

```
<Router> // Alle Routen müssen Teil des Routers sein,
      typischerweise nahe der Root-Komponente
<Route exact path="/" component={Home} /> // Component
//Home wird nur gerendert, wenn der path (exakt) matcht,
//Mehrere Route Elemente können gleichzeitig aktiv sein
<Link to="/">Home</Link> // App-interne Links verwenden
      nicht <a> sondern <Link>
```

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

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.

Effect Hook:

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

2.7 Typechecking

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

TypeScript:
- 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

2.8 Redux

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

Redux Actions:
Benötigt um Stateänderungen zu machen. Wird an den Store gesendet/dispatched. Ist eine reine Beschreibung der Action.

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

Redux Reducer-Funktionen:
Reducer sind pure Funktionen, haben also 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-Trees, 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 });
```

Store erstellen:
Mit dem root-Reducer kann der Store erstellt werden:

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

2.9 Redux mit React verbinden

mapStateToProps: Erhält State und kann daraus Props ableiten. Die Komponente bekommt auch die dispatch Methode des Stores als Prop. Das Resultat von connect ist wieder eine React-Komponente, die nun aber mit dem Store verbunden ist (Connected Component)
→ Store muss der Root-Komponente mitgegeben werden.
→ **Redux Thunk** erlaubt es uns, anstelle eines Objektes eine Funktion zu dispatchen

```
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') )
```

2.9.1 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}); }) }; }
```

3 Angular

3.1 Test

4 ASP.NET

4.1 Test