

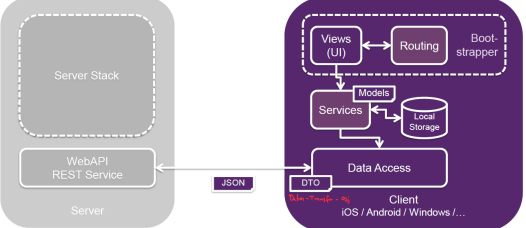
Introduction SPA
Browser-based Applications
Benefits
<ul style="list-style-type: none"> • Work from anywhere, anytime • Platform independent, including mobile • No software update, no application, easy maintenance • Software can be provided as a service (SaaS - pay as you go) • Code separation
Liabilities
<ul style="list-style-type: none"> • No data sovereignty (Datenhoheit) • Limited/restricted hardware access • SEO - Search engines must execute JavaScript • More complex deployment strategies

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.

Architecture

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



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.

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.

Routing
<ul style="list-style-type: none"> • Completely on client-side by JS • Navigation behaves as usual • Browser needs to fake the URL to change and store page state • <i>window.history.pushState</i>
Dependency Injection
Benefits
<ul style="list-style-type: none"> • Reduces coupling between consumer and implementation • Contracts between classes are based on interfaces • Supports the open/closed principle • Allows flexible replacement of an implementation

Decorators
<ul style="list-style-type: none"> • 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

React
<ul style="list-style-type: none"> • Library, kein Framework • Um User Interfaces zu bauen • View in MVC • Minimales Featureset • Entwickelt von Facebook • Verwendet für: WhatsApp, Insta, AirBnb, etc.

Prinzipien
<ul style="list-style-type: none"> • Komplexes Problem aufteilen in einfachere Komponenten • Für eine bessere: Wiederverwendbarkeit, Erweiterbarkeit, Wartbarkeit, Testbarkeit, Aufgabenverteilung

Entwicklung von UIs
<ul style="list-style-type: none"> • Beschreibung des UIs • Event-Handling • Aktualisieren der Views
Komponenten und Elemente
<ul style="list-style-type: none"> • Funktionen die HTML zurückgeben • Beliebige Komposition von React-Elementen und DOM-Elementen

```
function App() {
  return (
    <div>
      <HelloMessage name="HSR"/>
      
    </div>
  )
}
```

Parameterübergabe an Funktion

Äquivalent zu Attribut für DOM-Element

JavaScript XML
<p>React verwendet JSX (blau), eine Erweiterung von JavaScript (gelb). Überall wo JSX verwendet wird, muss <i>react</i> importiert werden.</p> <pre>const menu = entries.map(entry => <ListItem as="a" to={ `/\${entry.path}` }> <h1>{entry.title.toUpperCase()}</h1> <p>{entry.subtitle}</p> </ListItem>)</pre>
<p>Styles: werden nicht als Strings sondern als Object angegeben.</p>

Conditionals
<pre><Container> { error && <Message> fehler: {error} oder </Message> } </Container></pre> <pre><Container> ? error ? Fehler: {error} : OK! </Container></pre>

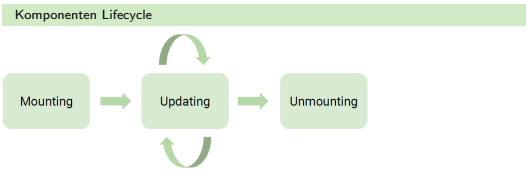
Props
<p>Komponenten erhalten alle Parameter/Properties als props Objekt.</p> <ul style="list-style-type: none"> • <i>this.props</i> bei Klassen • Bei Funktionen als Parameter • Immer read-only
Rendering und Mounting

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

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

React State
<p>React-Klassenkomponenten können einen veränderbaren Zustand haben. Der state einer Komponente ist immer privat. Ändert der State, wird auch die Komponente aktualisiert.</p> <pre>class Counter extends React.Component { state = { counter: 0 } // ... }</pre>
Event Handler
<pre>const increment = () => { this.setState({counter: this.state.counter + 1}) } // ... <button onClick={this.increment}></pre>
Reconciliation
<ol style="list-style-type: none"> 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

Formulare
<pre><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(); //... }</pre>
Komponenten Lifecycle



Mounting
<ol style="list-style-type: none"> 1. constructor(props) <ul style="list-style-type: none"> State initialisieren, sonst weglassen 2. static getDerivedStateFromProps(props, state) <ul style="list-style-type: none"> Von State abhängige Props initialisieren 3. render() 4. componentDidMount() <ul style="list-style-type: none"> DOM ist aufgebaut Guter Punkt um zum Beispiel Async-Daten zu laden setState Aufruf führt zu re-rendering
Updating
<ol style="list-style-type: none"> 1. static getDerivedStateFromProps(props, state) <ul style="list-style-type: none"> Von State abhängige Props aktualisieren 2. shouldComponentUpdate(nextProps, nextState) <ul style="list-style-type: none"> wird false zurückgegeben wird render übersprungen 3. render() 4. getSnapshotBeforeUpdate(prevProps, prevState) 5. componentDidUpdate(prevProps, prevState, snapshot) <ul style="list-style-type: none"> Analog zu componentDidMount, DOM ist aktualisiert
Unmounting
<ol style="list-style-type: none"> 1. componentWillUnmount() <ul style="list-style-type: none"> Aufräumen
Error Handling
<ol style="list-style-type: none"> 1. static getDerivedStateFromError(error) <ul style="list-style-type: none"> Error im State abbilden 2. componentDidCatch(error, info) <ul style="list-style-type: none"> Logging Verhindern, dass Fehler propagiert wird, analog zu catch-Block in try-catch
React Router
<ul style="list-style-type: none"> • Komponentenbibliothek • Komponenten anzeigen oder verstecken abhängig von der URL • Für React Web und React Native
Router Komponenten
<p><Router></p> <p>Alle Routen müssen Teil des Routers sein, typischerweise nahe der Root-Komponente</p> <p><Route exact path="/" component={Home} /></p> <p>Home-Komponente wird nur gerendert, wenn der path (exakt) matcht. Mehrere Route Elemente können gleichzeitig aktiv sein.</p> <p><Link to="/">Home</Link></p> <p>App-interne Links, welche nicht wie <a >die Seite neu laden.</p> <p><Redirect to="/somewhere/else"></p> <p>Wird ausgeführt, sobald gerendert.</p>
Hooks
<p>Problem von Lifecycle Methoden Zusammengehörender Code ist auf mehrere Methoden verteilt (Mount/Unmount).</p> <p>Problem von Klassen-State State ist über verschiedene Methoden verteilt</p> <p>Fazit:</p> <ul style="list-style-type: none"> • Lifecycle und State ohne Klassen machen react verständlicher • Klassen sind weiterhin unterstützt • Hooks erlauben, Logik mit Zustand einfacher wiederzuverwenden
State Hook
<pre>function Counter() { const [count, setCount] = useState(0); // button => setCount(count + 1) return(<p>{count}</p>); }</pre> <p>Mehrere State-Variablen: useState Aufrufe müssen immer in derselben Reihenfolge gemacht werden.</p>
Effect Hook
<pre>useEffect(() => { // Mount stuff return () => { // Unmount stuff } }, []) /* <= Dependencies */;</pre>
Flow
<ul style="list-style-type: none"> • Erweitert JavaScript um Typenannotationen • Typ-Annotation im Code Typ-Inferenz für lokale Definitionen • Generics, Maybe-Types, Union and Intersection-Types

TypeScript und React
<ul style="list-style-type: none"> • Mehr Typensicherheit in React-Komponenten • Props und State lassen sich typisieren <p>Vorteil gegenüber Flow:</p> <ul style="list-style-type: none"> • Vollwertige Programmiersprache • Besser unterstützt von Libraries und IDEs • TypeScript Fehler müssen korrigiert werden
React Context
<p>Ermöglicht es, Props für alle Unterkomponenten zur Verfügung zu stellen. (Theme Variablen)</p> <pre>// provider const c = React.createContext(themes.light); const theme = useContext(c); // consumer</pre>
Redux
<p>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.</p> <p>Actions</p> <p>Benötigt um Stateänderungen zu machen. Wird an den Store gesendet / dispatched. Action ist eine reine Beschreibung der Action.</p> <pre>{type: 'TRANSFER', amount: 100 }</pre>
Reducer
<p>Pure Funktionen, haben keine Seiteneffekte.</p> <pre>function balance(state = 0, action) { switch (action.type) { case 'TRANSFER': return (state + action.amount); default: return state; } }</pre> <p>Reducer kombinieren: Jeder Reducer erhält einen Teil des States, für den er zuständig ist. Resultat wird in einem neuen State-Objekt kombiniert.</p> <pre>function rootReducer(state = {}, action) { return { balance: balance(state.balance, action), transactions: transactions(state.transactions, action) } }</pre> <p>// Hilfsfunktion combineReducers:</p> <pre>const rootReducer = combineReducers({ balance, transactions });</pre> <p>Store erstellen</p> <pre>const store = createStore(rootReducer);</pre> <p>Mit dem root-Reducer kann der Store erstellt werden. In Kombination mit React führt das zu einem re-rendering der Komponenten.</p>
React <3 Redux
<p>Redux mit React verbinden:</p> <pre>const mapStateToProps = (state) => { return { transactions: state.transactions } }</pre> <pre>const mapDispatchToProps = { fetchTransactions }</pre> <pre>export default connect(mapStateToProps, mapDispatchToProps)(Component);</pre> <p>// Root Komponente</p> <pre>const store = createStore(rootReducer, applyMiddleware(thunkMiddleware)); render(<Provider store={store}> <App /> </Provider> document.getElementById('root'))</pre> <p><i>mapStateToProps:</i> erhält State und kann daraus Props ableiten. Die Komponente bekommt auch die <i>dispatch</i> Methode des Stores als Prop. Das Resultat von <i>connect</i> ist eine React-Komponente die mit dem Store verbunden ist.</p> <p>Store muss der Root-Komponente mitgegeben werden.</p> <p><i>thunkMiddleware:</i> Erlaubt es, anstelle eines Objektes eine Funktion zu dispatchen (benötigt für asynchrone Actions).</p> <p>Thunk Actions</p> <pre>function fetchTransactions(token) { return (dispatch, getState) => { dispatch({type: "FETCH_TRANSACTIONS_STARTED"}); api.getTransactions(token) .then(({result: transactions}) => { dispatch({type: " FETCH_TRANSACTIONS_SUCCEEDED", transactions}); }) }; }</pre>

Data Access

HTTP Client API

Implements asynchronisms by using the RxJS library. RxJS is a third-party library that implements the Observable pattern. An Observable can be turned into a promise.

Hot Observables: Sequences of events (mouse moves / stock tickers). Shared among all subscribers. Postfix hot-observables with a \$

Cold Observables: Start running on subscriptions (such as async web requests). Not shared among subscribers. Are automatically closed after Task is finished.

```
var subscription = this.http.get('api/samples').subscribe(  
  function (x) { /* onNext -> data received (in x) */ },  
  function (e) { /* onError -> the error (e) has been thrown */ },  
  function () { /* onComplete -> the stream is closing down */ }  
);
```

Routing

External, optional NgModule called RouterModule. Combination of multiple provided services and directives: *RouterOutlet*, *RouterLink*, *RouterLinkActive*.

Defining Routes: The router must be configured with a list of route definitions. Each definition maps a route to a component.

- *forRoot()*: use exactly once to declare routes on root level
 - contains all the directives, the given routes and the router service itself
 - Every app has one singleton instance of the router
- *forChild()*: When declaring sub-routings
 - contains all directives and the given routes

Each NgModule defines its own routes. Load modules on-demand (lazy load) with the *import*-Syntax.

```
@NgModule({  
  imports: [ RouterModule.forRoot(appRoutes) ], @NgModule({  
    imports: [ RouterModule.forChild(welcomeRoutes) ],  
    exports: [ RouterModule ]  
  })  
})  
export class AppRoutingModule {} export class WelcomeRoutingModule {}
```

Router Outlet: Directive from the Router module. Defines where the Router should display the views.

```
<router-outlet></router-outlet>
```

Route Configuration:

```
const appRoutes: Routes = [  
  // matches /hero/42, 42 saved in param  
  {path: 'hero/:id', component: 'Hero'},  
  // redirect  
  {path: '', redirectTo: '/heroes', pathMatch: 'full'},  
  // Wildcard route  
  {path: '**', component: PageNotFound}  
];
```

The router uses a first-match-wins strategy.

Lazy Loading Configuration

```
{ path: 'config',  
  loadChildren: () => import('./cfg/cfg.module').then(m => m.CfgModule),  
  canActivate: [ AuthGuard ] }
```

Angular Architectures

MVC+S

Observable Business Data Service: Provides data to multiple parts of the app in a stream-like manner. An *Observable* is provided. Stores/Caches business objects.

RxJS Subject: Heart of an observable data service. *EventEmitterT2* derives from Subject. Hot Observable and does not provide the latest value.

Behaviour Subject: Emits the initial state. Can be called some kind of warm. Stores the data and emits *next()* events on change. Do not expose to the Service API.

Data Resources: Return cold Observables. Must be converted into a hot Observable (*share()*).

Observable Business Data Service Example:

```
@Injectable({providedIn: 'root'})  
export class SampleService {  
  private samples: BehaviorSubject<SampleModel[]> = new BehaviorSubject([ ] );  
  public samples$: Observable<SampleModel[]> = this.samples.asObservable();  
  // Event bus which is used to store the last state and to notify subscribers about updates.
```

```
  // Postfix hot-observables (streams) with a $  
  // Convert event bus into an observable, which can be provided to the UI or other services.
```

```
  constructor(  
    private resourceService: SampleResourceService) {  
  }  
  // Call data resource service to store the data into the underlying backend-server.
```

```
  public addSample(newSample: SampleModel): Observable<any> {  
    return this.resourceService  
      .post(newSample)  
      .pipe(  
        tap(res => {  
          // Store the retrieved data into the BehaviorSubject and emit the data changed event. It is important to create a new array with the new data; otherwise, the async pipe won't track the change.  
          this.samples.next([...this.samples.getValue(), newSample]);  
        }  
      ),  
        catchError((err) => this.handleError(err)) );  
  }  
  private handleError(err: HttpErrorResponse): Observable<any> {  
    return new ErrorObservable(err.message);  
  }  
}
```

Flux Architecture

Invented by Facebook. Enforces a unidirectional data flow. More of a pattern than a formal framework.

Action creators are helper methods, collected into a library, that create an action from method parameters, assign it a type and provide it to the dispatcher.



Redux Architecture

ngrx: implements the Redux pattern using RxJS. **Benefits:**

- Enhanced debugging, testability and maintainability
- Undo/redo can be implemented easily
- Reduced code in Angular Components

Liabilities:

- Additional 3rd party library required
- More complex architecture
- Lower cohesion, global state may contain UI / business data
- Data logic may be fragmented into multiple effects/reducers

UI Advanced

Pipes

Can be applied within a template expression to make small transformations.

```
<p>{{counter.team | uppercase}}</p>  
<p>{{counter.team | uppercase | lowercase}}</p>  
<p>{{counter.date | date: 'longDate'}}</p>
```

Pure-Pipes: Executed when it detects a pure change to the input expression. Implemented as pure functions. Restricted but fast.

Impure-Pipes: Executed on every component change detection cycle (every keystroke etc.). To reduce processing time, caching is often used.

Predefined-Pipes: *date*, *number*, *currency*, *async* etc.

Angular does not provide Filter- / OrderBy-Pipes because of poor performance.

Custom-Pipes: A class decorated with *@Pipe()*. It implements the *PipeTransform* interface's *transform()* method. Needs to be added to the declarations of the current Module.

```

```

```
@Pipe({name: 'logo', pure: true})  
export class LogoPipe implements PipeTransform {  
  private logos = { /*...*/ };  
  transform(value?: string, transformSettings?: string): string {  
    if (value && transformSettings && this.logos[value]) {  
      return (this.logos[value][transformSettings] || this.logos[value].unspec);  
    }  
    return value;  
  }  
}
```

Async Pipes: Binds Observables directly to the UI. Changes are automatically tracked. Automatically subscribes and unsubscribes from the bound Observable.

```
<div>{{ sampleComponent | hi }}</div>  
<section>  
  <li *ngFor="let s of sampleService.samples$ | async">  
    <ul>{{s.name}}</ul>  
  </li>  
</section>
```

View Encapsulation

Component Styles: Apps are styled with standart CSS. The CLI transpiles SCSS to CSS. The selectors of a component's styles apply only within this own template.

Special Selectors:

- *:host* - Target styles in the element that hosts the component
- *:host-context* - Looks for a CSS class in any ancestor of the host element

Link Styles to Component - Options:

- Add a styles array property to the @Component decorator

```
@Component({ /*...*/  
  styles: ['h1 { font-weight: normal; }']  
})  
export class WedNavComponent { }
```
- Add styleUrls attribute into a components @Component decorator

```
@Component({ /*...*/  
  styleUrls: ['app/nav.component.css']  
})  
export class WedNavComponent { }
```
- Template inline tags/styles

```
@Component({ /*...*/  
  template: `<style>...</style> <link rel="stylesheet" href="app/nav.component.css">`  
})  
export class WedNavComponent { }
```

Controlling View Encapsulation:

- *Native:* Uses the browsers native shadow DOM
- *Emulated:* Emulates the behaviour of shadow DOM by preprocessing (and renaming) the CSS
- *None:* No view encapsulation (scope rules) applied. All CSS added to the global styles.