# Single Page Applications

**SPA Definition:** Eine Website, die auf einer einzigen Webseite Platz findet, mit dem Ziel, dem Benutzer ein ähnliches Erlebnis wie eine Desktop-Anwendung zu bieten. Verwendet AJAX und HTML5, um reaktionsfähige Web-Apps zu erstellen, ohne Seite ständig neu geladen wird.

|  |  |
| --- | --- |
| Vorteil | Nachteil |
| Funktioniert von überall (Flexibilität) | Keine Datensouveränität |
| Plattformunabhängig | Limitierte Hardwarezugriff (weniger effizient) |
| Keine Softwareupdates | Komplexere Deployment Strategien |
| Code Separierung | Overhead |
| Software as a service |  |

**Traditional Architecture:** Jedes Mal, wenn die Anwendung den Server aufruft, rendert der Server eine neue HTML-Seite

|  |  |  |
| --- | --- | --- |
| **Client** | ─Initial Request→ | **Server** |
| ←HTML─ |
| ─Form Post→ |
| **Page Reload**! | ←HTML─ |

**SPA Architecture:** Website interagiert mit user durch Änderungen von Teilen des DOM.

|  |  |  |
| --- | --- | --- |
| **Client** | ─Initial Request→ | **Server** |
| ←HTML─ |
| ─AJAX→ |
| ←JSON─ |

**Vorteile SPA und traditionelle Architektur:**

|  |  |
| --- | --- |
| SPA | Traditionelle Architektur |
| Schnelle Benutzererfahrung (Einmal laden dann nur noch dynamisch) | Suchmaschinenoptimierung (SEO) (Static HTML) |
| Interaktive Benutzeroberfläche (Keine Seitenladung mehr) | Einfachere Entwicklung und Debugging |
| Bessere Unterstützung für mobile Geräte (Reduzieren Daten Transfer) | Kompatibilität mit älteren Browsern |
| Effiziente Ressourcennutzung (Caching) |  |

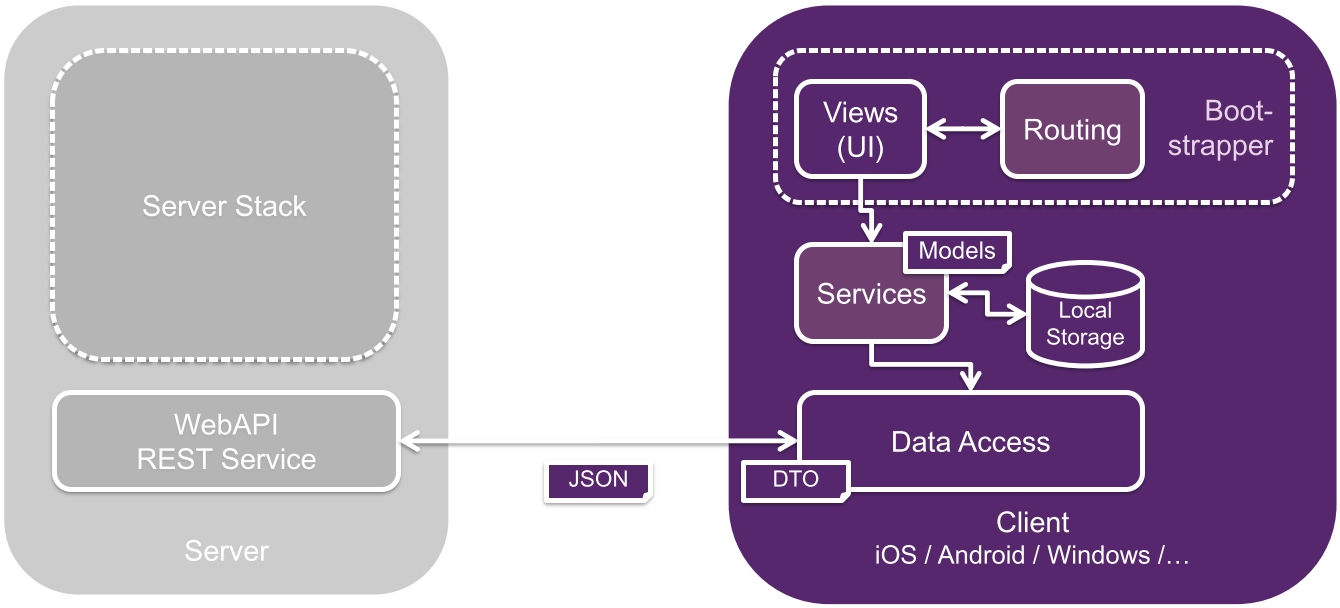
## architektur

**Characteristics:** Plain HTML5/CSS und JS, Keine Page Reloads, Funktionierender Back Button, Links als Bookmarks, Offline-Funktionalität, RESTful Services für Datenzugriff

**Logical Overview:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | |  | | --- | | **View Routing** | | **Models Services** | | **Data Access** | | **SPA** | | **Bundling** |

**SPA Architecture mit Layer:**

****

**Bundling SPA:** Gesamte JS-Code muss über potenziell langsame Netzwerke an den Client geliefert werden. Bündelung und Verkleinerung des Quellcodes führt zu einem kleineren SPA-Footprint. Größere SPAs mit vielen Modulen benötigen ein zuverlässiges Abhängigkeitsmanagement. Der anfängliche Fußabdruck kann reduziert werden, indem abhängige Module bei Bedarf geladen werden.

**WebPack als Bundler:**

* **Entry:** Bestimmt, wo das Bundling beginnen soll und folgt dem Graphen der Abhängigkeiten, um zu wissen, was gebündelt werden soll
* **Output:** Teilt WebPack mit, wohin die Anwendung gebündelt wird
* **Loaders:** Loader (Modul/Regel-API) in WebPack wandeln diese Dateien in Module um, wenn sie dem Abhängigkeitsgraphen hinzugefügt werden.
* **Plugins:** Loader führen Transformationen auf Dateibasis aus, Plugins werden am häufigsten für Aktionen und benutzerdefinierte Funktionen verwendet. Z.b Optimierung von Bundle, Verwaltung von Assets
* **Mode:** Integrierte Optimierungsmechanismen

## routing

Verlinkt mehrere Applikationsteile zusammen, um separate Applikationen in einen zusammenzuführen. Bietet das Konzept der Informationsarchitektur.

**Basic Principles:** Kein Page Reload bedeutet keinen Server Round-trip, nur Server’s Data-API ist involviert. Impliziert, dass Routing komplett Client-side ist. Funktionierender Back Button bedeutet, dass Navigation wie normal funktioniert, aber nur betroffener Content wird neu geladen. Impliziert, dass Browser die URL faken muss und die Page speichern muss.

**Technische Lösungen:** Mit HTML5 wird die JavaScript API window.history verwendet. window.history.pushState bewirkt, dass die Adressleiste eine gewünschte URL anzeigt, ABER nicht, dass der Browser sie lädt.

**Routing Implementation:** Einstiegspunkt (View (UI)) wird durch die angegebene Route erzwungen. Controller stellt Funktionen hinter einer View (UI) bereit und bootstrapst sie. Router bietet clientseitige Ereignis-Hooks während der Navigation (Lifecycle Management)

|  |  |  |
| --- | --- | --- |
| SPA | SPA | Routing Table |
| |  | | --- | | Route: /  Provide page overwiew, link  to /calendar | | |  | | --- | | Route: /calendar  Calendar View | | |  | | --- | | / :Overview  /calendar : CalendarView | |

# react

React ist eine Library (kein Framework) um UI zu bauen (V in MVC). Fokus auf Wartbarkeit, Wiederverwendbarkeit, Erweiterbarkeit und Aufgabenverteilung im Team.

**Aufgaben bei der Entwicklung von UI:**

1. Beschreibung der Struktur des UI: Programmier- oder Markupsprache (Java Swing, HTML, XAML, Android-XML)
2. Event-Handling: Listener und Callbacks registrieren
3. Aktualisierung der View: Observer an-/abmelden (evtl. Data Binding), Views ein-/ausblenden, Übergänge zu anderen Views/Pages

**Npx:** Node.js command-line tool, welches die Ausführung von command-line tools erlaubt, ohne diese global zu installieren (Projekte brauchen unterschiedliche Versionen desselben Packages oder Packages werden nicht oft gebraucht)

**JavaScript XML (JSX):** JSX ist eine Erweiterung von JavaScript. In {} stehen JavaScript-Expressions.

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

**Einschränkungen:** React-Elemente müssen mit Grossbuchstaben anfangen. JavaScript-Keywords können nicht verwendet werden (bsp. class wird zu className)

return (<div className="container">{props.children}</div>)

**Children:** XML-Unterlemente werden als children Array in den Props übergeben. Wenn man JSX schreibt, muss man die HTML-Tags verwenden.

**→** Grossschreibung = Reactkomponente / Kleingeschrieben = HTML-Element

function Container(props) {  
 return (<div className="cont">{props.children}</div>)  
}  
  
function *App*() {  
 return (<Container><HelloMessage name="OST"/> </Container>)  
}

**Styles:** Style-Attribute werden nicht als String übergeben, sondern als Objekt. Die JavaScript DOM Style-Objekte verwenden camelCase (z.B. minHeight).

function Container(props) {  
 const style = {  
 display: 'flex',  
 width: '100 %',  
 minHeight: 300, *// Keine Einheit = px*  
 };  
 return <div style={style}>{props.children}</div>;  
}

**Conditionals:** Elemente können bedingt gerendert werden. Was zu null, true, false oder undefined evaluiert wird nicht ausgegeben.

|  |  |
| --- | --- |
| <Container>   {error &&   <Message>   Fehler: {error}  </Message>}  </Container> | <Container>  {false}  </Container> |
| <Container>  <Message>  Fehler: {error}  </Message> </Container> |

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

**Desugaring:** JSX wird vom Präprozessor zu React.createElement Aufrufen gewandelt. React muss deshalb immer importiert werden wenn JSX verwendet wird

|  |  |
| --- | --- |
| JSX | JS |
| import React from 'react'  function Container(props) {  return <div className="c">  {props.children}  </div> } | import React from 'react'  function Container(props) {  return React.*createElement*("div",   {className: "c"},   props.children) } |

**Komponenten:** Als Funktion, Lambda oder Klasse möglich. Klassenkomponenten wurden von Hooks abgelöst, aber noch nicht deprecated

function Msg(props) { *// Funktion*  
 return <div>Hello {props.name}</div>;   
}

const Msg = (props) => <div>Hello {props.name}</div>; *// Lambda*

class HelloMessage extends React.Component { *// Klasse*  
 render() { *// Props sind nun ein Attribut* return <div>Hello {this.props.name}</div>  
 }}

**Props:** Komponenten erhalten alle Parameter/Properties als props Objekt. Props sind immer Objekte und immer als READ-ONLY zu behandeln.

function Container({children, ...props}) {  
 return <div className="container" {...props}>{children} </div>  
}   
  
<Container style={{textAlign: "center"}}>  
 <h1>Hello WE3</h1>  
</Container>  
<div className="container" style={{textAlign: "center"}}>  
 <h1>Hello WE3</h1>  
</div>

**Rendering und Mounting:** Um Komponenten auf einer Webseite anzuzeigen müssen wir diese mounten. Dazu benötigen wir die render Methode eines Root-Elements.

<body>   
 <div id="root"></div>  
</body>

import React from 'react';  
import ReactDOM from 'react-dom/client';  
import App from './App';

const r = ReactDOM.*createRoot*(  
 document.getElementById('root'));

root.render(<App/>);

**State:** Web-Applikationen sind dynamisch und interaktiv. React-Komponenten können einen veränderbaren Zustand haben. In React lassen sich diese Zustände mit einem State abbilden.

**useState Hook:** Mit der useState Funktion können wir in unserem Komponenten State erfassen. useState ist ein erster sogenannter Hook. Wert ist sofort verfügbar (kein Getter)

function Counter() {

*// Wert Setter Initialwert*  
 const [counter, setCounter] = *useState*(0);  
 const increment = () => setCounter(counter + 1);  
 return (<button onClick={increment}>{counter}</button>)  
}

Bei State-Hooks mit mehreren Variablen. useState Aufrufe müssen immer in derselben Reihenfolge gemacht werden, nur so kann React den State verwalten (keine if)

function LoginForm() {  
 const [firstname, setFirstname] = *useState*("");  
 const [lastname, setLastname] = *useState*("");  
}

**Event Handler:** Um den Zustand zu ändern benötigen wir einen Event-Handler. React-Events erwarten eine Funktion als Parameter, entweder inline oder mit Hilfe einer Funktion.

function increment() { setCounter(counter + 1) };   
<button onClick={increment}>Increment Counter</button>

**Regeln für State:**

* State einer Komponente ist immer privat. Kann aber als Prop (!) und damit read-only an andere Komponente weitergegeben werden
* Auch Event-Handler/Setter können als Props an Komponenten weitergegeben werden
* Zustand darf ausschliesslich mit Settern geändert werden
* Props bevorzugen, nur wenn nötig State nehmen
* Keine von Props abgeleiteten Daten im State speichern!

**Reconciliation:** React-Komponenten werden als virtueller DOM gerendert. Wird der state geändert, erstellt React einen neuen virtuellen DOM. Alter und neuer DOM werden verglichen und erst dann werden geänderte DOM-Knoten im Browser erstellt.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | verändert |  | aktualisiert |  |
| App |  | Virtual DOM |  | DOM |
|  | Events |  | Events |  |

**Formular:** Event Handler bei den Inputs registrieren und Zustand ändern. Jede Eingabe des Benutzers wird nun unmittelbar im username State abgelegt und über den value dargestellt. Man spricht deshalb auch von Controlled Components. Werden die Formulardaten vom DOM selbst verwaltet und nicht in React-State abgelegt, so ist dies Uncontrolled Component

<input value={username} type="text" onChange={e => setUsername(e.target.value)} />

**Formular Submission:** Zwei Varianten: onClick des Buttons, oder onSubmit des Formulars. Da man in einer SPA per Ajax-Call kommunizieren soll, will man keinen POST Request haben. => event.preventDefault()(wen nicht vorhanden sendet der Browser das Formular ab)

function handleSubmit(event) {  
 event.preventDefault();  
 *alert*("User: " + username + ", Password: " + password)  
}

**Virtual DOM:** Virtuelle DOM ist eine abstrakte Repräsentation des tatsächlichen DOM, die von der Browser-Engine verwaltet wird. Für Performanceoptimierungen und batched updates.

**Session Storage:** Daten als Key/Value im Browser ablegen. Dabei ist zu beachten, dass es sich die Lebenszeit der Daten im Session Storage auf eine Session beschränken. Wird ein Fenster/Tab geschlossen, das den Session Storage benützt, gehen ebenfalls alle Daten im Session Storage verloren. Für eine längere Lebenszeit Local Storage verwenden

**Widget-Libraries:** Es gibt einige Libraries welche sich um das Styling kümmern wie Reactstrap, Material-UI, Atlaskit etc.

## Komponenten Lifecycle

**Lifecycles:** React Klassen-Komponenten haben 3 Lifecycle-Methoden

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 zum bsp. Async-Daten zu laden. setState Aufruf führt zu re-rendering
2. **Updating (Wiederholt sich):**
   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
3. **Unmounting:**
   1. componentWillUnmount(): Aufräumen

**Error Handling:**

* static getDerivedStateFromError(error): Error im State abbilden
* componentDidCatch(error, info): Logging, Verhindern dass Fehler propagiert wird analog zu catch von try-catch

**Beispiel:**

class Clock extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {date: new Date()};  
 }  
 componentDidMount() {  
 this.timerID = *setInterval*(this.tick, 1000)  
 }  
 componentWillUnmount() { *clearInterval*(this.timerID)}  
 tick = () => { this.setState({date: new Date()}) }  
 render() {  
 return (  
 <p>It {this.state.date.toLocaleTimeString()} </p>  
 );  
 }  
}  
ReactDOM.render(<Clock/>, document.getElementById('root'));

**Probleme von Lifecycle-Methoden:** Zusammengehörender Code ist auf mehrere Methoden verteilt. Kann unübersichtlich und dadurch fehleranfällig sein. Wiederverwendbarkeit solcher Logik ist kaum möglich. Braucht Klassen.

**Beispiel mit Effekt Hook:**

useEffect(() => { *// Wird bei mount ausgeführt* *// Function body*  
 return () => { *// Wird bei unmount ausgeführt*

*// Function body*

}

}, [])

Als zweiten Parameter nimmt useEffect ein Array von Dependencies entgegen. Ändert sich eine der Dependencies, wird der Effekt erneut ausgeführt.

**useEffect für API-Zugriffe:**

const [remoteData, setRemoteData] = *useState*(null);  
const [error, setError] = *useState*(null);  
const [isLoading, setIsLoading] = *useState*(false);  
*useEffect*(() => {  
 setIsLoading(true); *// Loading-Indicator anzeigen*  
 *fetch*('https://jsonplaceholder.typicode.com/todos/1')  
 .then((response) => {  
 if (!response.ok) { throw new *Error*('Oops'); }  
 return response.json();  
 })  
 .then((data) => setRemoteData(data))  
 .catch((error) => setError(error.message));  
 setIsLoading(false); *// Loading-Indicator ausblenden*  
}, [todoId]); *// Bei Änderung wird Funktion neu ausgeführt.*

**Typechecking:** Als JavaScript-Library ist React natürlich nicht statisch typisiert. Flow (lib) erweitert JavaScript um Typannotationen. Typ-Annotation im Code Typ-Inferenz für lokale Definitionen.

function add(num1: number, num2: number) { return num1 + num2 }  
const x = add(3, '0') *// Incompatible types error*

Nach einem Null-Check weiss Flow, dass x nicht null ist und berücksichtigt das im Verlauf

function length(x) { if (x != null) {return x.length;} }

**Typescript:** Mehr Typsicherheit in React-Komponenten. Props und State lassen sich typisieren. Für React und viele Bibliotheken stehen Typdefinitionen zur Verfügung

## react router

Library, welche nicht Teil von React ist, um Komponenten anzuzeigen oder verstecken anhand der URL. Für React Web und Native

* <BrowserRouter>: Alle Routen müssen Teil des Routers sein. Typischerweise nahe der Root-Komponente
* <Route path="/about" element ={<About/>} />: Component About wird nur gerendert, wenn der path matcht. Routes können auch verschachtelt werden
* <Link to="/about">About</Link>: App-interne Links verwenden nicht <a> sondern <Link>

**Beispiel:**

function App() {  
 return (<BrowserRouter>  
 <div>  
 <Link to="/">Home</Link>  
 <Link to="/about">About</Link>  
 <Routes>  
 <Route index element={<Home/>}/>  
 <Route path="/about" element={<About/>}/>  
 </Routes>  
 </div>  
 </BrowserRouter>) }

## react context

**Problem:** Daten als Props mitzugeben ist mühsam, Zustand und Calls zum Backend über ganze Applikation verteilt.

**React Context:** React Context ermöglicht es, Props für alle Unterkomponenten zur Verfügung zu stellen. Bsp. Theme

**Context Provider:** Die App kann einen Context erstellen und über einen Provider die Variablen im Context setzen.

|  |  |
| --- | --- |
| const ThemeContext =  React.*createContext*(themes.light); function App() {  return (  <ThemeContext.Provider value={themes.dark}>  <Toolbar/>  </ThemeContext.Provider> );} | const themes = {  light: {*…*},  dark: {*…*}, }; |

**Context Consumer:** Alle Unterkomponenten von ThemeContext.Provider können mit dem useContext Hook nun auf die Werte zugreifen.

const theme = *useContext*(ThemeContext);

**Context Fazit:** Dank Context können Props an beliebig tief verschachtelte Kinder weitergegeben werden. Ideal für “globale” read-only Props wie Themes. Wird bei vielen verschiedenen Context unübersichtlich.

# redux

Library für Statemanagement, Repräsentation/ Veränderungen des States, Benachrichtigung bei Änderungen. Erlaubt die Möglichkeit für Server-side rendering.

**Redux State:** State wird als Tree von Objekten dargestellt, welcher immutable ist. Veränderungen am Tree führen zu einem neuen Tree (Funktionale Programmierung). State wird im sogenannten Store verwaltet, der benutzt Reducer-Funktion um neuen State zu erstellen

**Store:** Verwaltet den State. Invoked by actions

**Redux Action:** Um eine Stateänderung zu machen brauchen wir eine Action. Action wird an den Store gesendet. Action ist eine reine Beschreibung der Action

{ type: 'TRANSFER', amount: 100 }

**Redux Reducer:** Store benutzt Reducer-Funktion, um einen neuen State-Tree anhand der Daten in der Action zu erstellen. Reducer sind pure Funktionen, haben also keine Seiteneffekte.

|  |  |  |
| --- | --- | --- |
| State t | Reducer | State t+1 |
| Balance: 200 | Type: „TRANSFER“  Amount: 100 | Balance: 300 |

function balance(state = 0, action) {  
 switch (action.type) {  
 case 'TRANSFER': *// Neuer Wert für State t+1* return (state + action.amount)   
 default: *//Actions interessieren diesen Reducer nicht* return state   
 }  
}

**Reducer kombinieren:** Ein Reducer ist immer nur für einen Teil (Slice) des State-Trees zuständig. Anderer State hat einen eigenen Reducer. Bsp. transactions für Transactions. Mehrere Reducer werden dann zu einem Root-Reducer kombiniert. Inline State für die App ist ein leeres Objekt. Jeder reducer erhält nur den Teil (Slice) des States, für den er zuständig ist. Resultat wird zu einem neuen State-Objekt kombiniert.

function rootReducer(state = {}, action) {  
 return {  
 balance: balance(state.balance, action),   
 transactions: transactions(state.transactions, action)  
 }  
}

Um das zusammenstellen zu erleichtern gibt es eine Hilfsfunktion:

const rootReducer = combineReducers({ balance, transactions })

**Store erstellen:** Mit dem (root) Reducer können wir nur den Store erstellen

const store = createStore(rootReducer);

**Listener:** Konzept, um über Änderungen im State benachrichtigt zu werden

store.subscribe(() => console.log(store.getState()));

In Kombination mit React führt das dann zu einem re-rendering der Komponenten, dazu später mehr.

## react redux toolkit

React Container-Komponenten müssen Zugriff auf den Redux-State haben - Mapping von Redux-State nach React-Props, Actions an den Redux-Store dispatchen können und bei Änderungen am State müssen Komponenten neu gerendert werden.

**Probleme:** Bei der Benutzung von plain Redux treten immer wieder ähnliche Komplikationen auf: Konfiguration des Stores zu kompliziert (connect, mapStateToProps), Viele nützliche Dev Tools müssen separat hinzugefügt werden, Zu viel Boilerplate-Code

**Redux Toolkit:** Standard für die Redux Entwicklung. Verschiedene Features erlauben die schnelle und einfache Integration des Redux Stores und die Generierung von neuen States, Reducern und Actions. Die grundlegende Logik von Redux bleibt dieselbe. Das Toolkit schafft somit ein weiteres Abstraktionslevel für die vereinfachte Umsetzung des State Managements mit Redux.

**createSlice:** createSlice erstellt neue Stateobjekte, Reduce-Funktionen und Aktionen in einem.

const balanceSlice = *createSlice*({  
 name: "balance",  
 initialState: {value: 0},  
 reducers: {  
 transfer: (state, action) => {  
 state.value += action.payload.amount;  
 },  
 },  
});  
export const {transfer} = balanceSlice.actions;

Scheinbar direkte Änderungen am State möglich (Copy and Update Strategie). Jedoch Änderung am State durch Kopieren und Update. createSlice (und createReducer) nutzen Immer.js.

**configureStore:** Initialisiert den Redux Store mit den angegeben Reducern. Default Initialisierung enthält redux-thunk um asynchrone Actions (Promises) zu dispatchen und Redux Dev Tools.

import {*configureStore*} from "@reduxjs/toolkit";  
import {balanceReducer} from "./redux/balanceSlice";  
  
const store = *configureStore*({ reducer: {balance: balanceReducer} });  
export default store;

**Provider:** Um den Store für connect innerhalb der React-Komponentenhierarchie verfügbar zu machen müssen wir diesen unserer Root-Komponente noch mitgeben

import store from './store';

render(  
 <Provider store={store}>   
 <App/>  
 </Provider>, document.getElementById('root') )

**useDispatch:** Wird für den Dispatch der Aktionen an den Store benutzt. Die Aktionen führen die Reducer-Funktion auf dem Statetree aus.

import {*useDispatch*} from "react-redux";  
const dispatch = *useDispatch*()  
dispatch(*transfer*({amount: 10}))

**useSelector:** Wird für die Abfrage des States benutzt. Wie eine getter-Funktion

import {useSelector} from "react-redux";  
const balance = useSelector(state => state.balance.value);

**Asynchrone Actions:** Actions sind einfache Objekte, die über den Store dispatched werden.

Redux Thunk erlaubt es uns, anstelle eines Objektes eine Funktion zu dispatchen. Diese Funktion kann selbst wieder Actions dispatchen, z.B. nach einem fetch. Thunk ist eine sogenannte Middleware welche durch die configureStore bereits konfiguriert.

import {createAsyncThunk, *createSlice*} from '@reduxjs/toolkit';  
import api from "./api";  
*// First, create the thunk export*const tAsync = createAsyncThunk("balance/transferApiRequest",  
 async (amount) => {  
 const response = await api.transfer(amount);  
 return response.data;  
 });

dispatch(tAsync({ amount: 10 }))

**createSlice with asyncThunk:**

const balanceSlice = *createSlice*({  
 initialState: {value: 0, status: "idle"},

extraReducers: (builder) => {  
 builder.addCase(tAsync.pending, (state) => {  
 state.status = "loading";  
 }).addCase(tAsync.fulfilled, (state, action) => {  
 state.status = "idle";  
 state.value += action.payload.amount;  
 });  
 },  
});

**Beispiel:**

import { *createStore* } from 'redux'  
  
function counterReducer(state = { value: 0 }, action) {  
 switch (action.type) {  
 case 'counter/incremented':  
 return { value: state.value + 1 }  
 case 'counter/decremented':  
 return { value: state.value - 1 }  
 default:  
 return state  
 }  
}  
*// Create a Redux store holding the state of your app.  
// Its API is { subscribe, dispatch, getState }.*let store = *createStore*(counterReducer)  
  
*// You can use subscribe() to update the UI in response to state changes.  
// Normally you'd use a view binding library (e.g. React Redux) rather than subscribe() directly.  
// There may be additional use cases where it's helpful to subscribe as well.*store.subscribe(() => console.log(store.getState()))  
  
*// The only way to mutate the internal state is to dispatch an action.  
// The actions can be serialized, logged or stored and later replayed.*store.dispatch({ type: 'counter/incremented' }) *// {value: 1}*store.dispatch({ type: 'counter/incremented' }) *// {value: 2}*store.dispatch({ type: 'counter/decremented' }) *// {value: 1}*

## sonstige libraries

**JHipster:** Generataor für komplexe Applikationen (Angular, React-Redux, Vue etc.), User Management, CRUD

|  |  |
| --- | --- |
| Vorteile | Nachteile |
| Super schneller Start, kaum Zeitverlust durch Setup von Build, CI/CD, etc. | Round-Trip-Engineering nicht unterstützt |
| Viele mühselige CRUD-Programmierarbeit wird einem abgenommen | Neugenerierung nach Änderung an JDL ist möglich, aber Änderungen müssen von Hand gemerged werden |
| Mit JDL erhält man in wenigen Minuten einen ersten Prototypen einer Applikation | User Interface nutzt Bootstrap, leider kein mächtigeres Toolkit |

**Jest:** Unit Testing von React Komponenten. Bereits Teil von mit create-react-app erstellten Applikationen. Stellt render, screen, etc. Utilities bereit, um Komponenten ohne Browser zu testen.

import {*render*, screen} from '@testing-library/react'  
import userEvent from '@testing-library/user-event'  
  
*test*('loads and displays greeting', async () => {  
 *// ARRANGE  
 render*(<Fetch url="/greeting"/>);  
 *// ACT* await userEvent.*click*(screen.*getByText*('Load Greeting'));  
 await screen.*findByRole*('heading');  
 *// ASSERT  
 expect*(screen.*getByRole*('heading')).toHaveTextContent('hello there');  
 *expect*(screen.*getByRole*('button')).toBeDisabled();  
})

# angular

Angular ist eine flexible SPA Framework für CRUD-Applikationen. Kommt mit Dependency Injection Mechanismus. Stellt ein schnelles 2-way binding bereit. Erhöht Testbarkeit und Wartbarkeit von Client-side code. Angular bietet eine strikte Struktur und enthält viele Funktionen, während React flexibler ist und sich auf die UI-Komponenten-Darstellung konzentriert.

**Architektur:**

* **ngModules:** Ein zusammenhängender Codeblock, der eng miteinander verbundenen Fähigkeiten gewidmet ist. (Typescript module)
* **Directives:** Instruktionen um das DOM zu transformieren (Typescript class)
* **Components:** Directive-with-a-template. Kontrolliert eine Sektion des Views (Typescript class)
* **Templates:** Eine Form HTML welcher Angular sagt wie die Komponente gerendert wird (HTML, CSS)
* **Metadata:** Beschreibt eine Klasse und sagt wie Angular dies verarbeiten soll. „Schmiermittel“ zwischen Template und Component (Typescript decorators)
* **Services:** Liefert die Logik eines beliebigen Wertes, einer Funktion oder eines Merkmals, die die Anwendung benötigt. (Typescript class)

## module

Angular-Modul ist eine logische Einheit, die Komponenten, Direktiven, Pipes und Services bündelt, um den Code in Anwendungen zu organisieren und strukturieren. Definiert die Funktionalität und Abhängigkeiten. Jede Angular-Anwendung hat ein Hauptmodul namens AppModule

**ngModule:** Verhalten sich anders als TS und ES6 Module. Ein ngModule ist ein logischer Block von mehreren TS Modulen zusammen gelinked. Die ngModule Deklaration wird in einem TS Modul platziert und kann ein „barrel“ zur Verfügung stellen, welches die public API exportiert.

import {*NgModule*} from '@angular/core';  
import {BrowserModule} from '@angular/platform-browser';  
@NgModule({  
 *// Module whose exported ate needed* imports: [BrowserModule],  
 *// Services this NgModule contributes to global collection of services;* providers: [Logger],  
 *// The components, directives, pipes belong to this NgModule.* declarations: [AppComponent],  
 *// subset of declarations that should be visible and usable* exports: [AppComponent],  
 *// Main view, called the root component, which hosts all the views. Only the root*

*// NgModule should set the bootstrap property.* bootstrap: [AppComponent]  
})  
export class AppModule {}

**Module Declaration:**

* **declarations:** [ Type1, Type2, Type3, …]: View Klassen welche zu diesem Modul gehören. Angular kennt 3 Arten (Components, Directives, Pips)
* **exports:** [ Type1, Type2, … Module1, Module2 ]: Subset von Deklarationen welche sichtbar und verwendbar sein sollen in den Component templates von anderen Modulen. Kann andere Module erneut exportieren.
* **imports:** [ Module1, Module2, …]: Spezifiziert Module welche Exports/Providers in diesem Modul importiert werden müssen
* **providers:** [ Provider1, Provider2, …]: Ersteller von Diensten, die dieses Modul in die globale Sammlung von Services (Dependency Injection Container) einbringt; sie werden in allen Teilen der Anwendung zugänglich.
* **bootstrap:** [ Component ]: Die Hauptansicht der Anwendung, die so genannte root component. Nur das Wurzelmodul sollte diese Eigenschaft einstellen.

## components

Komponenten sind die grundlegenden Bausteine einer Angular-Anwendung. Sie bestehen aus einer TypeScript-Klasse, einer HTML-Datei (Template) und optionalen CSS-Stilregeln. Komponenten sind verantwortlich für die Darstellung und Logik einer bestimmten Ansicht innerhalb der Anwendung. (Component Tree Komponenten beinhalten andere Komponenten über den Selector im Template)

**Architektur:** Components erlaubt, UI-Logik von der Repräsentation zu separieren. Kann verschachtelt werden, was zu einem Component tree führt. Kann Informationen verbergen dadurch dass jede Komponente einen Teil der UI deklariert. Eine Komponente sollte als kleines zusammenhängendes Stück wie möglich implementiert werden, um Testbarkeit Wartbarkeit und Wiederverwendbarkeit zu unterstützen. Sind Teile von einem ngModule welcher üblicherweise mehrere Components definiert.

**Component Declaration:** Eine Komponente besitzt genau 1 View entweder mit template URL oder direkt inline definiert. Selector wird verwendet um Komponente in anderer View (templete) einzubetten. Optional kann ein CSS oder Inline-Style definiert werden.

import { *Component* } from '@angular/core';  
@Component({  
 selector: 'app-root',  
 templateUrl: './app.component.html',  
 styleUrls: ['./app.component.css'],  
})  
export class AppComponent { title = 'Tour of Heroes'; }

**Component Registration:** Komponenten müssen innerhalb des enthaltenen Moduls deklariert werden. Kann exportiert werden, damit andere Module diese importieren können.

@NgModule({  
 declarations: [NavigationComponent],   
 imports: [CommonModule],   
 exports: [NavigationComponent],   
 providers: []  
})  
export class SharedModule {}

**Component Usage:** Eine andere Ansicht kann einfach auf die Komponente verweisen, indem sie den Komponenten-Selektor anspricht

<router-outlet></router-outlet>

**Lifecycle:**

|  |  |
| --- | --- |
| constructor | **ngOnInit (hydration)**  erst hier sollten Serviceaufrufe gemacht werden (falls dieser Service z.B von einem Lazyload Service abhängt)  **ngOnDestroy (dehydration)**  sobald Component ausgeblendet wird |
| ngOnChanges |
| ngOnInit |
| ngDoCheck |
| ngAfterContentInit |
| ngAfterContentChecked |
| ngAfterViewInit |
| ngAfterViewChecked |
| ngOnDestroy |

**Content Transclusion/Content Projection:** Wiederverwendbar Angular component erlaubt Parametrisierung der View.

|  |  |
| --- | --- |
| *<!--component usage-->* <section>  <wed-navigation>  <h1 wedTitle>WED3<h1>  <menu>  *<!--…-->*  <menu>  </wed-navigation> </section> | *<!--navigation.component.html-->* <header>  <ng-content select='[wedTitle]'>  </ng-content> </header> <nav>  <ng-content select='menu'>  </ng-content> </nav> |
| *<!--resulting DOM-->* <section>  <wed-navigation>  <header> <h1 wedTitle>WED3</h1></header>  <nav> <menu>*<!-- ... -->*</menu> </nav>  </wed-navigation> </section> | |

## templates

Template ist bekannt als die View in MVC. Geschrieben in HTML und annotiert mit Angular template syntax. Sagt Angular wie die Komponente gerendert wird. Kann in andere Komponente eingebettet werden was in einen Komponentenbaum resultiert.

**Angular Template Syntax:** Praktisch alle HTML5 Syntax in valider template syntax (ausser script tag). Angular erweitert HTML um Interpolation, Statements/Expressions, Binding Syntax, Directives, Template Reference Variables und Template Expression Operators.

**Binding:** Mechanismus zur Koordinierung der Verwaltung von UI-abhängigen Informationen. Applikation ist einfacher zu schreiben und warten.

* **One-way in von Datenquelle zu View:** Property Binding  
  <p>... {{counter.team}} ..</p>   
  <img [attr.alt]="counter.team" src="team.jpg">
* **One-way call von View zu Datenquelle:** Event Binding  
  <button (click)="counter.eventHandler($event)">
* **Two-way:** Property and Update Event Binding  
  <input type="text" [(ngModel)]="counter.team">

**Interpolation:** Stellt eine Einwegbindung von der Datenquelle zum Ansichtsziel dar. Kann in Verbindung mit der Template Expression Syntax verwendet werden. Einige JS Expressions sind illegal wie ++, --, new, += , ,, etc.. Operatoren wie |, &, ?, ; haben eine andere Bedeutung. **Binding Input and Output properties:** Bindings an Ziele müssen als Eingänge oder Ausgänge deklariert werden. Targets stehen auf der linken Seite der binding declaration. Diese directive/component properties müssen als Input oder Output deklariert werden.

<wed-navigation (click)="..." [title]="..."> </wed-...>

* **@Output:** Input ermöglicht das Übergeben von Daten von übergeordneten zu untergeordneten Komponenten in Angular.
* **@Input:** Output erlaubt das Emittieren von Ereignissen von untergeordneten zu übergeordneten Komponenten in Angular.

export class ChildComponent {  
 @Input() item = '';  
 @Output() newItemEvent = new EventEmitter<string>();  
  
 addNewItem(value: string) {  
 this.newItemEvent.emit(value);  
 }  
}

*<!—html code-->*

<app-child [item]="string" (newItemEvent)="log($event)"/>  
*<!--log() must be defined in the Parent-->*

## directives

Ähnlich einer Komponente aber ohne template. Deklariert in TS mit einem @Directive().

**Attribute Directives:** Ändert das Aussehen oder Verhalten eines Elements, einer Komponente oder einer anderen directive. Wird als Attribut auf ein Host-Element angewendet.

**ngStyle:** Legt die Inline-Stile dynamisch fest, basierend auf dem Zustand der Komponente.

<div [ngStyle]="{'font-size': isSpecial ? 'x-large' : 'smaller'}"> *<!-- render element -->* </div>

**ngClass:** Bindet an die ngClass-Directive, um mehrere Klassen gleichzeitig hinzuzufügen oder zu entfernen.

<div [ngClass]="hasWarning ? 'warning' : '' "> *<!-- render element -->* </div>

**Structural Directives:** Verantwortlich für das HTML-Layout. Umgestaltung der DOM-Struktur, typischerweise durch Hinzufügen, Entfernen oder Manipulieren von Elementen. Anwendung auf ein Host-Element als Attribut. Ein Sternchen (\*) steht vor dem directive attribute name

**ngIf:** Nimmt einen boolean Wert an und lässt einen ganzen Teil des DOMs erscheinen oder verschwinden.

<div *\*ngIf*="hasTitle">*<!-- shown if title available -->*</div>

**ngFor:** Stellt eine Möglichkeit dar, eine Liste von Elementen zu präsentieren.  
<li *\*ngFor*="let element of elements">*<!--render shit-->*</li>

Asterisk ist syntactic sugar und Angular de-sugars directive in ein template.

<ng-template [ngIf]="hasTitle">  
 <div>*<!--conditional content-->*</div>  
</ng-template>

**Ng-template and:** Nicht direkt gerendert, sondern brauchen eine directive/component welches es übernimmt. Kann referenziert werden mit #id.

<ng-template #toReference>*<!-- content -->*</ng-template>

<div *\*ngIf*="hasTitle; else toReference">*<!-- conditional content -->*</div>

**Ng-container:** Ng-container werden immer gerendert und kann mit einem if kombiniert werden zum ausblenden.

**Template Reference Variables:** Referenziert DOM-Element innerhalb von einem template. Kann zu einem Angular component/directive referenzieren. Reference Variables können überall im template verwendet werden. # deklariert dies.

<input placeholder="phone number" #phone>  *<!-- phone refers to the input element; pass its `value` to an event handler -->*<button (click)="callPhone(phone.value)">Call</button>

**Beispiel:**

import {*Directive*, ElementRef} from '@angular/core';  
  
@Directive({  
 standalone: true,  
 selector: '[appHighlight]',  
})  
export class HighlightDirective {  
 constructor(private el: ElementRef) {  
 this.el.nativeElement.style.backgroundColor = 'yellow';  
 }  
}

## services

Service ist eine weit gefasste Kategorie, die jeden Wert, Funktion oder Merkmal umfasst, das eine Anwendung benötigt. Ein Service ist typischerweise eine Klasse mit eingem, klar definierten Zweck. Es sollte etwas Bestimmtes tun und das gut tun. Typische Services sind Logging, Data Service, Message Bus, Application Configuration etc.

**Service Declaration:** Füge den @Injectable()-Dekorator einer Serviceklasse hinzu, damit Angular diese als Abhängigkeit dependecy injecten kann. Optionales Argument teilt Angular mit, wo diese Klasse standardmäßig registriert werden soll.

@Injectable({providedIn: "root"})  
*// registered for whole application with tree-shaking  
// Ist nur im Bundle, wenn irgendwo referenziert*export class CounterService{…}

Alternativ kann der Service als Teil des ngModule Deklaration deklariert werden. Globale und Einzeln instanzierbare Dienste in forRoot() platzieren. Diese Option ist nützlich, wenn man multi-providers (e.g Validators) oder konfigurierbare Providers will.

**Data Services:** Data Resources repräsentiert Service von Data Access Layer. Solche Data Resource Services werden verwendet um HTTP zu enkapsulieren. Angular simplifiziert dies mit dem HttpClient. Http service providers können im Root Modul registriert werden mittels Import HttpClientModule.

**Hot Observables:** Sequenz von Events, Geteilt zwischen alle Subscriber

**Cold Observables:** Observables starten mit der Subscription, Nicht geteilt, Automatisch geschlossen nachdem Task fertig ist.

const subscription = this.http.get('api/samples').subscribe({  
 next: (x) => { */\* onNext -> data received (in x) \*/* },   
 error: (e) => { */\* onError -> error has been thrown \*/* },   
 complete: () => { */\* onCompleted -> stream closing down \*/* }  
});

## forms

Ein Angular-Formular koordiniert eine Reihe von datengebundenen user controls, verfolgt Änderungen, validiert Eingaben und zeigt Fehler an. Ist ein externes, optionales Angular NgModule namens FormsModule.

**Reactive (model-driven) forms:** Erfordert den Import des ModulsReactiveFormsModule. Formular wird innerhalb des Controllers erstellt (FormBuilder). Validierungslogik ist auch Teil des Controllers und daher einfacher zu testen. Kann mehrere asynchrone Validierungen verwalten.

**Template-driven forms:** Angular template syntax mit den formularspezifischen directives und techniques. Erzeugt weniger Code, platziert aber die Validierungslogik in HTML. Intuitiver, simpler, weniger JS/TS Code. Nützlich für kleine Formulare. Angular ngForm directive verbessert form tag mit erforderliche Funktionalitäten.

<form>  
 <input type="text" class="form" pattern="[a-zA-Z]{3,}">  
</form>

**Validation:** Wirdautomatisch angewandt. Referenziert auf die [ngModel] directive und überprüft obs gültig ist. Um directive (als Objekt) in der Vorlage zu referenzieren, wird eine spezielle Syntax verwendet: #nameField="ngModel"

<form>  
 <input type="text" class="form-control" id="name"  
 required [(ngModel)]="model.name" name="name"  
 #nameField="ngModel">  
 <div [hidden]="nameField.valid || nameField.pristine"  
 class="alert alert-danger">  
 Name is required Template  
 </div>  
</form>

**Submitting:** ngForm kann auch referenziert werden. Kann nützlich sein, um validation state auf den submit button zu binden.

<form (ngSubmit)="doLogin(sampleForm)" #sampleForm="ngForm">  
 <button type="submit" class="btn btn-success"  
 [disabled]="!sampleForm.form.valid">  
 Submit  
 </button>  
<form>

@Component({...})  
export class SampleComponent {  
 public doLogin(f?: NgForm): boolean {  
 if (f?.form.valid) { *// store data* }  
 return false; *// avoid postback* }  
}

## component lifecycle

Komponente besitzen eine Life Cycle, welche Entwickler darauf zugreifen können (z.B ngOnInit, ngOnDestroy, etc.). Um sich in den life cylce einzuklinken, können Schnittstellen des Angular-Core implementiert werden. Jedes Interface hat eine hook Methode, prefix ng. Angular ruft nur hook Methoden auf, da Interfaces vom TS compiler entfernt werden.

@Component({ … })   
export class CounterComponent implements OnInit, OnDestroy {  
 ngOnInit() { console.log("OnInit"); }   
 ngOnDestroy() { console.log("OnDestroy");}  
}

## asynchronous services

Services enthalten den Grossteil der Applikationslogik. Interagiert mit Data resources asynchron. Kann Möglichkeiten bieten um Daten local auf dem Client zu cachen.

**Kommunikation Service und UI:** Angular stellt viele Features zusammen mit RxJS dar. RxJS kann die Komplexität von asynchronen Datenmanagement verringern.

**Beispiel Asynchrone Service:**

@Injectable({providedIn: 'root'})  
export class SampleService {  
 private samples: SampleModel[] = []; *// simple cache  
 // Create emitter instance. The type argument specifies   
 // the kind of object to be passed to the subscriber.* public samplesChanged: EventEmitter<SampleModel[]> = new EventEmitter<SampleModel[]>();  
 constructor( */\* inject data resource service \*/*) {}  
  
 load(): void { */\* in real word app, invoke data resource service here \*/* this.samples = [new SampleModel()];  
 *// Logic to execute when data ready. Emit changed event to notify the  
 // registrars (e.g. UI components).* this.samplesChanged.emit(this.samples);  
 }  
}  
export class SampleModel {}

**Beispiel Asynchrone Service (Usage):**

@Component({...})  
export class SampleComponent implements OnInit, OnDestroy {  
 private samples: SampleModel[];  
 *// Subscription is used to unsubscribe the update event when the  
 // component is de-hydrated.* private samplesSubscription: Subscription;  
  
 constructor(private sampleService: SampleService) {}  
  
 ngOnInit() {  
 *// Register samplesChanged event on underlying business service when component is hydrated.  
 // Subscribe() returns a Subscription which is used for deregistration.* this.samplesSubscription = this.sampleService.samplesChanged.subscribe((data: SampleModel[]) => {  
 *// Update procedure; refresh data on the UI level.* this.samples = data;  
 });  
 }  
 ngOnDestroy() {  
 *// Unsubscribe the update event when the component is de-hydrated.* this.samplesSubscription.unsubscribe();  
 }  
}

## routing

Optionaler Angular Router um zwischen Views zu navigieren. Ist eine externe und optionale ngModule. Jedes ngModule sollte sein eigenes Routing Module enthalten. Initiale Top-level Routes müssen mit forRoot() registriert werden.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Root Module**  Routing Module  forRoot() | |  | | --- | | ‘\*\*’ => PageNotFoundComponent  ‘dashboard’ => lazy import(DashboardModule) | | | |
|  |  |  |  |
| import | **Weclome Module**  Routing Module  forChild() | |  | | --- | | ‘welcome’ => WelcomeComponents | | |
|  |  |  | |
| import | **Dashboard Module**  Routing Module  forChild() | |  | | --- | | ‘ ’ => DashboardComponent | | |

**Base href:** Angular verwendet history.pushState vom Browser für Navigation. HTML5 style Navigation ist Standard und kann von # Navigation überschrieben werden.

**forRoot():** forRoot()-import genau einmal verwendem um Routen im root zu definieren. Beinhaltet alle directives, Routen und den Router selbst. Eine Angular App hat einen Singleton Instanz des Router Service.

**forChild():** forChild()-import wenn Subrouten definiert werden. Routes von oben gegen unten definieren! Damit nicht pagenotfound immer gematched wird.

**Router Outlet:** RouterOutlet ist eine directive vom Router Module. Definiert wo Router die View anzeigen soll. Kann innerhalb einer child component definiert werden. Component wird dann in router-outlet einfügt.

{path: 'dashboard', component: DashboardComponent}

<a routerLink="/dashboard">Open Dashboard</a>  
<router-outlet></router-outlet>

**Route Configuration:** Router verwendet first match wins Strategie. D.h spezifischere Routes sollen über weniger spezifische Routes stehen. Ideal ist eine Wildcard route (\*\*) einzufügen um invalide URL abzufangen. pathMatch: ‚full‘ kann verwendet werden für einen exakten Match.

**Lazy Modules:** Lazy Modules werden geladen, sobald die Route das erste Mal angefragt wird. Lazy Module sollten nicht direkt von eagrly loaded Modules referenziert werden, sonst muss das Modul bei Applikationsstart verfügbar sein.

**Beispiel:**

const appRoutes: Routes = [  
 { path: 'crisis-center', component: CrisisListComponent },  
 *// Maps the /crisis-center path to the CrisisListComponent* { path: 'hero/:id', component: HeroDetailComponent },  
 *// Maps the /hero/42 path to the HeroDetailComponent "42" is variable,*

*// specifies the value of the id parameter; can be read out by code* { path: '', redirectTo: '/heroes', pathMatch: 'full' },  
 *// Redirects the default route (/) path to the /heroes path. '' must be*

*// exactly matched.* { path: '\*\*', component: PageNotFoundComponent }  
 *// Specifies a wildcard route (two asterisks) which matches every URL.  
 // Used if the requested URL doesn't match any of the defined routes.*];

**Beispiel child route:**

const routes: Routes = [  
 { path: 'samples', component: SamplesComponent,  
 *// Maps /samples path to the SamplesComponent . This component also  
 // contains a router outlet. http://localhost:4200/samples* children: [  
 { path: ':id', component: SamplesDetailComponent },  
 *// Defines the router outlet of SamplesComponent should be filled with the  
 // SamplesDetailComponent if an id has specified.  
 // http://localhost:4200/samples/42* { path: '', component: SamplesListComponent }  
 *// Defines the router outlet of SamplesComponent should be filled with the  
 // SamplesListComponent if no sub-path given. http://localhost:4200/samples/* ]  
 }  
];

## import mechanisms

**Default Import:** Importiert alle components, pipes, directives für ein Modul. Providers warden in aktuellen DI container registriert.

@NgModule( { imports: [ ForeignModule ] })

**forChild Import:** forChild(config?) repräsentiert eine statische Methode auf Modul. Fast gleich wie default import, aber erlaubt Services für den aktuellen Modullevel zu konfigurieren.

@NgModule( { imports: [ ForeignModule.*forChild*( { } ) ] })

**forRoot Import:** forRoot(config?) repräsentiert eine statische Methode auf Modul. Stellt und konfiguriert Service gleichzeitig. Nur Root Module ist es erlaubt forRoot auf fremde Module aufzurufen.  
@NgModule( { imports: [ ForeignModule.forRoot( { } ) ] })

## module types

* **Root/App Module:** Stellt Eingangspunkt für app zur Verfügung. Hostet alle anderen app views. Kein Export. Muss BrowserModule importieren. Globales Routing.
* **Core Module:** Hält Root Modul sauber. Enthält components/directives/pipes welche vom Root verwendet werden. Hier können auch global verwendete Services deklariert werden. Nur Root Module sollte Core Module importieren.
* **Shared Modules:** Enthält gemeinsame components/directives/pipes der App und teilt sie mit feature modules. Globale UI component module. Soll nicht ein app-wide singleton zur Verfügung stellen, eher lazy-loaded module welcher eine Kopie des Services erstellen kann. Singleton besser in Root Module definieren
* **Feature Modules:** Spezifiziert Grenzen zwischen Applikationsfeatures. Kann kategorisiert werden in detailliertere Modultypen.
  + **Domain:** UI dediziert zu einer Applikationsdomain
  + **Routing:** Routing spezifische Konfiguration des Feature- order Root-Modul
  + **Service:** Utility services wie data access und messaging
  + **Widget:** Macht components/directives/pipes verfügbar für externe Module

## mvc+s architecture

Angular bietet viele Features in Kombination mit RxJS. Es ist möglich, RxJS durch alle Schichten (UI / Service / Data Resource) zu ziehen. Führt am Ende zu asynchronen Abschnitten innerhalb der UI.

**Observable Types:**

* Hot Observables: Sequenz von Events (z.B Mouse Movement events oder wie ein Youtube video)
* Cold Observables: Observables starten beim subscriben and schliessen automatisch ab (bsp. Livestream)

**Observable Business Data Service:** Bereitstellung von Daten für mehrere Teile der Anwendung in einer stream-like Art. Dies bedeutet, dass eine Observable bereitgestellt wird. Speichert/zwischenspeichert business objects.

Das Herz eines Observable data serivce ist das RxJS Subject. Ein Subject repräsentiert einen traditionallen multicast event bus, aber stärker. Es ist ein hot Observable und stellt nicht den neusten Wert dar.

**Behavior Subject:** RxJS stellt BehaviorSubject zur Verfügung, welches das Problem oben löst. Emittiert den initialen Stand und dann next() bei Änderungen. Sollte nicht für Service-API freigegeben werden, da clients sonst next() aufrufen können.

**Accessing Data from Data Resources:** Data resources geben cold Observables zurück. Der Business Service und dessen Subscriber müssen über eine erfolgreiche Datenmutation informiert werden.

*// Event bus which is used to store last state and notify subscribers about updates.*private samples: BehaviorSubject<SampleModel[]> = new BehaviorSubject([]);  
*// Postfix hot-observables (streams) with a $.*public samples$: Observable<SampleModel[]> = this.samples.asObservable();  
  
*// Convert event bus into an observable, which can be provided to UI or services.*constructor(private resourceService: SampleResourceService) { }

## redux architectur

Viele Angular-Bibliotheken für reaktive Erweiterungen. ngrx implementiert das Redux-Muster unter Verwendung des bekannten RxJS.

Diagram

Description automatically generated

|  |  |
| --- | --- |
| **Benefits** | **Liabilities** |
| Verbessertes Debugging, Testbarkeit und Wartbarkeit | Zusätzliche Bibliothek eines Drittanbieters erforderlich (ngrx) |
| Undo/Redo kann einfach implementiert werden | Komplexere Architektur |
| Reduzierter Code in Angular Components | Geringere Kohäsion, "Global State" kann Geschäfts- und UI-Daten enthalten |
| Bekannter Datenfluss (Component -> Store -> Component) | Datenlogik kann in zahlreiche Effekte/Reduzierer aufgesplittert sein |
| Reduzierter Overhead bei der Änderungserkennung |  |

**Conclusion:** Vermischung von Pipes und Filtern mit Future Pattern ist eine schlechte Idee da Fehler schwer zu finden sind. Unintuitive Verwendung der Framework-API - rtm first!

## pipes

Transformation vom Resultat einer Expression, anwendbar in einer template expression. Das Ergebnis eines Ausdrucks kann eine Transformation erfordern. Gut für kleine Transformationen. Können gekettet werden.

<p>{{counter.date | date:'longDate'}}</p>*<!--May 25, 1970-->*

* **Pure Pipes (default):** Werden ausgeführt, wenn eine "pure change" (Funktionen ohne Seiteneffekte) des Eingabeausdrucks feststellt wird. Restriktiv, aber schnell
* **Impure Pipes:** Werden bei jedem Zyklus von component change ausgeführt. Können Seiteneffekte beinhalten (Funktionen die bei gleichen Input verschiedene Werte zurückgeben können). Das bedeutet jeder Tastendruck und Mausklick.

**Predefined Pipes:**

* **AsyncPipe:** Impure, unwraps value from async value observable/promise | async
* **DecimalPipe** Pure,formats number to locale rules | number[:digitsInfo[:locale]]
* **DatePipe** Pure, Uses internationalization API date\_expr | date[:format[:timezone]]
* **PercentPipe** Pure, Formats number as percentage according to locale rules | percent[:digitsInfo[:locale]]

**Custom Pipes:** Eine Pipe ist eine Klasse mit einer Pipe Metadata Annotation (@Pipe). Implementiert die PipeTransform interface’s transform() Methode.

<img src=“{{ counter?.team | logo:‘toImage‘}}“>

@Pipe({name: 'logo', pure: true})  
export class LogoPipe implements PipeTransform {  
 *// Specific arguments tob e passed to the transform() method* private logos = {*/\*...\*/*};  
  
 transform(value ?: string, transformSettings ?: string): string {  
 *//…* return value;  
 }  
}

**Async Pipes:** Observables können direkt auf die UI gebunden werden mit AsyncPipe. Änderungen werden automatisch getracked. Pipe kann automatisch subscriben und unsubscriben vom gebundenen Observable. Um Änderungen zu tracken, soll AsyncPipe neue Werte/Referenzen vom Observable erhalten.

|  |  |
| --- | --- |
| <h1>WE3</h1>  <section>  <li \*ngFor=``let s of saService.samples$ | async``>  <ul>{{s.name}}</ul>  </li>  </section> | @Component({…})  export class CounterComponent{  constructor(  public saService: SaService  ){}  …. |

# pwa and more

PWA nutzen Web-API mit Progessive Enhancement um cross-plattform Web Apps zu erstellen.

* **Safe:** Die neuen Features wie Push-API sind nur über HTTPS möglich. Chrome erlaubt diese Features auch auf Localhost für die Entwicklung.
* **Responsive:** Das UI soll für alle Grössen passen. Wie Media Queries. (Desktop, Tablet, Smartphone, Brillen, Uhren)
* **Progressive:** Die neuen API’s nutzen um die bestmögliche UX zu erreichen, ohne die älteren Browser zu ignorieren 🡪 Progressive Enhancement
* **Linkable:** Teilen von Inhalten ohne das der Sharing-Partner eine App installieren muss.
* **Discoverable:** Die Webseite soll über Suchmaschinen auffindbar sein 🡪 Web App Manifest
* **Installable:** Die Web-Applikation mit Icon auf den Startbildschirm speichern 🡪 Web App Manifest
* **Network independent:** Die App soll mit schlechter, langsamer oder sogar gar keiner Verbindung funktionieren.
* **Re-engageable:** Es soll möglich sein das die Verbindung zum Benutzer der Seite wieder aufgenommen werden, obwohl der Benutzer die Seite nicht geöffnet hat. Service Workers, Push-API

**Firebase:** Backend Services für Mobile/Web (Build, Release&Monitor, Engage)

**Cypress:** E2E Tests. JHipster generiert End-to-End Tests

**MobX:** Alternative zu MobX. Write minimalistic, boilerplate free code that captures your intent. …, the reactivity system will detect all your changes and propagate them out to where they are being used.

|  |  |
| --- | --- |
| Redux | MobX |
| Single Store | Multi Store |
| Plain Objects | Business Objects |
| Immutable Store | Mutable Store |
| Normalized Data | Denormalized Data |
| Unidirectional data flow | Unidirectional data flow (recommended) |

# asp.net

**Convention over configuration:** Unterstützt Entwickler beim Treffen von Entscheidungen, um die Entwicklung mit dem Framework zu vereinfachen.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| HTTP Request | **Entry point**  Front Controller  (does routing) |  |  |  |  |
|  |  |  |  |  |  |
|  | **Controller** |  | **Model** |  | **Data Source**  (database, Api) |
|  |  |  |  |  |  |
| HTTP Response | **View** |  |  |  |  |

**MVVM:** View (UI), View Model (UI-Logik), Model (Daten, Services)

**Attribute:** Für Klassen/Methoden/Properties um Konventionen zu überschreiben

[Required]  
[StringLength(maximumLength: 50, ErrorMessage = "Rip")]  
public string Street { get; set; }

**Multi-Threading:** ASP.NET besitzt einen konfigurierbaren Thread Pool, für jeden Request wird ein Thread aus dem Pool genommen. Thread blockiert solange bis Request abgeschlossen ist. Möglichkeit Thread zurückzugeben mit async/await

**Front Controller:** Entry Point eines http Request, ist für Routing zuständig.

**Middleware:** Ein Request durchläuft ein Stack von Middlewares. Jede Middleware kann den Request beenden (Bsp. Autosierung, Logging, MVC, WelcomePage, …)

var builder = WebApplication.*CreateBuilder*(args);var app = builder.Build(); *// Add services to the container.*app.Use(async (context, next) => {*// Neue Middleware registrieren*  
 await context.Response.WriteAsync("Hello");  
 await next.Invoke();  
});

app.Map("/logging", builder => { *// Verzweigung für Anfragepfad*  
 builder.Run(async (context) => {

await context.Response.WriteAsync("Hello");

});  
});

*// Terminiert den Request, keine weiteren Middlewares werden aufgerufen*  
app.Run(async (context) => {   
 await context.Response.WriteAsync("World");  
});

**Dependency Injection:** Reduziert Kopplung zwischen verschiedenen Klassen. Verbesserung der Testbarkeit einer Applikation. Eine Klasse erwähnt welche Interfaces benötigt werden. Ein Resolver sucht im Container nach einer geeigneten Klasse und übergibt diese. Im Container wurden die Interfaces mit der konkreten Klasse registriert. Falls keine konkrete Klasse vorhanden ist, wird üblicherweise ein Fehler geworfen.

var builder = WebApplication.CreateBuilder(args);  
*// Add services to the container.*builder.Services.AddTransient<IUserService, UserService>();  
var app = builder.Build();  
*// Configure the HTTP request pipeline.*app.UseMiddleware<UserMiddleware>();  
app.Run();

**Dependency Injection – Lifeftime:**

|  |  |
| --- | --- |
| Transient | Wird jedes Mal bei einem Request erstellt. Bestens für lightweight, stateless Services |
| Scoped | Service wird einmal pro Request erstellt |
| Singleton | Service wird nur einmal erstellt und nachfolgende Requests brauchen die gleiche Instanz |

## pages

Alternative und vereinfachte Variante vom MVC, Router muss nicht konfiguriert werden. Best-Practices für Serverseitiges-Rendering.

**Projekt-Struktur:**

|\_wwwroot: Alle statischen Inhalte der Webpage (CSS/JS/HTML)

|\_appsettings.json: Die Einstellungen der Webseite z.B. Connection-String zur DB

|\_Programm.cs: Einstiegspunkt, Middleware und Dependency Injection Konfiguration

**Routing:**

1. Eine Web Applikation generiert anhand von der URL eine Antwort.

2. Eine URL wird auf eine Aktion "gemappt".

3. Bei einem Aufruf wird im Folder /pages/ nach einer Page gesucht und ausgeführt. Parameter definiert überschreibt die Default-Routing-Informationen (/order/add -> pages/order/add.cshtml)

**MVVM:** Pages hat 2 Files, \*.cshtml View mit Razor geschrieben und \*.cshtml.cs ist View Model. View Model kann pro HTTP-Verb eine Funktion definieren die davor aufgerufen wird (OnGet, OnPost, …)

|  |  |
| --- | --- |
| @page  @model HelloWorldModel  @{ViewData["Title"] = "Hello";} <h1>@Model.HelloWorld</h1> | public class HelloWorldModel : PageModel {  public string HelloWorld {get;set;}  public void OnGet() {  HelloWorld = "Hi World!";  } } |

**Razor:** Template Engine mit C# ähnlicher Syntax. Das @ wechselt zwischen HTML und C# Code

@{ var total = 7; } *<!-- Single statement blocks -->*  
*<!-- Inline expressions -->*<p>The value of your account is: @total </p>   
@{ *<!-- Multi-statement block -->*  
 var greeting = "Welcome to our site!";   
 var weekDay = DateTime.Now.DayOfWeek;   
 var greetingMessage = greeting + " Today is: " + weekDay;  
}  
<p>The greeting is: @greetingMessage</p>

**Unobtrusive Ajax:** Query Unobtrusive AJAX"-Library ermöglicht es Form-Requests ohne eigenen JavaScript Code zu definieren

## validation

Client-Side Validation mit bsp. JQuery und Server-Side Validation mit ASP.NET.

1. Annotieren der Klassen:

StringLength(MinimumLength = 8)]   
[Required]  
public string Password { get; set; }

1. Validierung im DOM:
2. Server-Side Validierung

[HttpPost] public ActionResult Index(Order order) {  
 if (ModelState.IsValid) {  
 order.CustomerId = User.Identity.GetUserId();   
 \_db.Orders.Add(order);   
 \_db.SaveChanges();   
 return View("OrderOk", order);  
 }   
 return BadRequest();  
}

## rest api

Funktioniert über Attribute. [Route] definiert einen neuen Eintrag im Router. [HttpMethod] bei Actions ist required.

[Route("api/[controller]")] public class ValuesController : Controller {  
 [HttpGet]   
 public IEnumerable<Value> Get() {  
 return \_valueService.All();

}  
 [HttpGet("{id}")]   
 public Value Get(int id) {  
 return \_valueService.Get(id);

}  
 [HttpPost]   
 public void Post([FromBody]Value value) {  
 \_valueService.Add(value);  
 }   
}

**Location:** Location-Header sollte korrekt gesetzt sein.

public IActionResult Post([FromBody]Value value) {  
 value = \_valueService.Add(value);   
 return new CreatedAtActionResult("Get", "Values", new {id = \_valueService.GetId(value)}, value);  
}

**Swagger:** Dokumentataion von REST-API. Programmiersprachen unabhängig. Basiert auf Open API Beschreibungsstandard. Muss im Program.cs eingetragen werden.

## exception handling

Error Handling soll generisch funktionieren. Es gibt eine Exception, die notwendigen Daten sammelt. Es gibt einen globalen Errorhandler, welcher diese Exception für Client aufbereitet. Bei einem ungültigen Zustand wird Custom-Exception ausgelöst. Es gibt eine Middleware, welche die Exceptions fangen und verarbeiten kann

public class ServiceException : Exception {  
 public ServiceExceptionType Type { get; private set; }  
 public ServiceException(ServiceExceptionType type) {   
 Type = type;  
 }  
}  
public Value Get(int id) {   
 if (Values.Count > id && Values[id] != null) {   
 return Values[id]; }   
 throw new ServiceException(ExceptionType.NotFound);  
}

## authentifizierung

**Authentifizierung:** Identifikation vom Nutzer (Passwort,Handy, Biometrie). Typische Probleme werden vom Framework abgenommen wie Passwort-Stärke, User Validator, Lockout Mechanismus, Two Faktor Authentifikation, SMS, User und Role Management, Passwort zurücksetzen, OAUTH Anbindung

|  |  |
| --- | --- |
| UserManager<ApplicationUser> | Verantwortlich für die Verwaltung von Benutzer |
| RoleManager<IdentityRole> | Verantwortlich für die Verwalten von Rollen |
| IAuthorizationService | Verantwortlich für die Validation von Policies |
| SignInManager | Verantwortlich für das Einloggen  User + Password / 2FA / OpenId Connect, etc. |

Controller und Pages können [Authorize] besitzen, die nur zugegriffen werden können wenn eingeloggt. This.User beinhaltet den eingeloggten User, das ein ClaimsPrincipal ist.

[Authorize]   
public ActionResult Create() {  
 return View(new Order() { Name = "Hawaii" });  
}

**Claim:** Statement über einen User. Ausgestellt von einem Identity Provider (User1 hat die Admin role, User1 hat Telefonnummer «216541654651651»).

**Autorisierung:** Ist was man darf.

1. Mit Annotation:   
   [Authorize(Roles = "Admin,PowerUser")]

[Authorize(Policy = "OlderThan18,Founders")]

1. Services

var user = await \_userM.GetUserAsync(User);

var inRole = await \_userM.IsInRoleAsync(user, "Admin");  
var result = await \_authService.AuthorizeAsync(User, null, "Founders");

1. Claims  
   User.HasClaim(ClaimTypes.Role, "Admin")

**JWT-Token:** Standard zum Tokens zu generieren zum eine Nummer von Claims zu verifizieren. Beinhaltet Header, Payload (Claims), Signatur. Es braucht ein Zertifikat oder Key für die Signierung / Validierung der Token, ein Endpoint, Erstellung JWT-Token und Middleware für Token Validation (z.B JwtBearerMiddleware). Tokens sollte nur über HTTPS verschickt werden.

## random stuff

**Integration Tests:** Testen, ob die Applikation im Ganzen funktioniert. Üblicherweise werden Scenarios getestet. Z.B. Anmelden, in den Warenkorb legen, bestellen, ausloggen. Integration-Tests können einen die Web-Applikation starten und diesen für die Tests benutzen.

**Gründe View Models für REST-API zu definieren:**

* **Security:** Klare Definition wie Daten im Backend aussehen sollen und welche ans Frontend gesendet werden
* **Information Hiding:** Zusätzliche Abstraktion erlaubt API-Definition von Business Logik und DB Schema zu trennen.

**Gemeinsamkeiten Angular und .Net:** MVC-Architektur, Typesicherheit, Data Binding, DI

**Vorteil DI:** Reduziert Kopplung zwischen Konsumenten und Implementierung, Verträge zwischen Klassen basieren auf Schnittstellen, Unterstützt das Open/Closed-Prinzip, Ermöglicht flexiblen Austausch einer Implementierung

# web assembly / blazor

WebAssembly hat weitreichende Auswirkungen auf die Webplattform. Es bietet eine Möglichkeit, Code, der in mehreren Sprachen geschrieben ist, auf der Webplattform mit nahezu nativer Geschwindigkeit auszuführen. Dadurch können clientseitige Anwendungen im Web ausgeführt werden, die zuvor dies nicht gekonnt hätten.

**WAT:** Text Based Web Assembly. Kann zu WAM kompiliert werden

**WASM:** Compiled Web Assembly. Kann zu WAT zurücktransformiert werden.

**Modul:** Repräsentiert ein WebAssembly Binary welches vom Browser in ein executable kompiliert wurde.

**Memory:** Ein Veränderbarer ArrayBuffer, der das lineare Array von Bytes enthält, die von den Low-Level-Speicherzugriffsanweisungen von WebAssembly gelesen und geschrieben werden. Kann in JS oder WASM alloziert werden.

**Table:** Ein veränderbares typisiertes Array von Verweisen (z.B. auf Funktionen), die sonst nicht als raw Bytes im Speicher abgelegt werden könnten (aus Sicherheit- und Portabilitätsgründen)

**Instance:** Modul gepaart mit dem State welches es bei Runtime verwendet mit Memory, Table und set von importierten Werten.

**Blazor:**

|  |  |
| --- | --- |
| **Blazor Server** | **Blazor Web Assembly** |
| Statische Webseite mit etwas Interaktion  Server-side Rendering (SSR)  Schnelle Entwicklung  keine Daten-API benötigt | SPA Lösung |

Blazor Server: Islands of interactivity

Statische Seiten mit wenig Interaktion z.B. Produkt Seiten von einem Webshop

|  |  |  |
| --- | --- | --- |
| **Features Blazor** | **Server** | **WebAssembly** |
| Complete .NET API compatibility | ✔️ | ❌ |
| Direct access to server and network resources | ✔️ | ❌ |
| Small payload size with fast initial load time | ✔️ | ❌ |
| Near native execution speed | ✔️ | ✔️ |
| App code secure and private on the server | ✔️ | ❌ |
| Run apps offline once downloaded | ❌ | ✔️ |
| Static site hosting | ❌ | ✔️ |
| Offloads processing to clients | ❌ | ✔️ |
| Full access to native client capabilities | ❌ | ❌ |
| Web-based deployment | ✔️ | ✔️ |

# code

## react

**Dashboard.js:**

const *Dashboard* = ({token, isAuthenticated, user}) => {  
 const navigate = *useNavigate*();  
 const dispatch = *useDispatch*();  
  
 const [amount, setAmount] = *useState*(undefined);  
 const transactions = *useSelector*((state) => state.transaction.allTransactions);  
  
 const isValidTargetAccount = (accountNr) => {  
 return *getAccount*(accountNr, token).then(  
 () => true,  
 () => false  
 );  
 };  
  
 const handleSubmit = (target, amount) => {  
 *transfer*(target, amount, token).then(() => {  
 *// Transfer succeeded, we just re-fetch the account details  
 // instead of calculating the balance ourselves  
 getAccountDetails*(token).then(({amount}) => {  
 setAmount(amount);  
 });  
 *// same for the transactions  
 getTransactions*(token).then(({result: transaction}) => {  
 dispatch(addTransactionToState({allTransactions: transaction}))  
 }  
 );  
 });  
 };  
 *useEffect*(() => {  
 if (token) {  
 *getTransactions*(token).then(({result: transaction}) => {  
 dispatch(addTransactionToState({allTransactions: transaction}))  
 }  
 );  
 }  
 }, [token, transactions, dispatch]);  
  
 return (  
 <Container>

*/\* Some HTML codoe \*/*  
 </Container>  
 );  
};  
export default *Dashboard*;

**loginSlice.js:**

import {*createSlice*} from "@reduxjs/toolkit";  
  
export const loginSlice = *createSlice*({  
 name: "login",  
 initialState: {  
 user: {},  
 token: "",  
 isAuthenticated: false  
 },  
 reducers: {  
 addUserToState: (state, action) => {  
 state.user = action.payload.user;  
 state.token = action.payload.token;  
 state.isAuthenticated = true;  
 },  
 removeUserFromState: (state) => {  
 state.user = {};  
 state.token = '';  
 state.isAuthenticated = false;  
 },  
 },  
});  
  
export const {addUserToState, removeUserFromState} = loginSlice.actions;  
export const loginReducer = loginSlice.reducer;

## angular

**App.module.ts:**

@NgModule({  
 declarations: [AppComponent],  
 imports: [  
 DashboardModule,  
 CoreModule.*forRoot*(),  
 AuthModule.*forRoot*(),  
  
 AppRoutingModule  
 ],  
 providers: [  
 {provide: LOCALE\_ID, useValue: 'de-CH'}  
 ],  
 bootstrap: [AppComponent]  
})  
  
export class AppModule { *// RootModule* constructor() {  
 *registerLocaleData*(localeDeCH, 'de-CH', localeDeCHExtra);  
 }  
}

**Dashboard.component.ts:**

@Component({  
 selector: 'wed-dashboard',  
 templateUrl: './dashboard.component.html',  
 styleUrls: ['./dashboard.component.scss']  
})  
export class DashboardComponent {  
 public target?: number;  
 public amount?: number;  
  
 constructor(private accountService: AccountService) {} *// DI*

onRecipientChange() { *// Logic* }  
  
 ngOnInit() {  
 this.accountService.account.subscribe((data: Account) => {  
 this.recipientAccount = data;  
 });  
 }

}

**App-routing.module.ts:**

const appRoutes: Routes = [  
 {  
 path: 'dashboard',  
 loadChildren: () => import("./dashboard/dashboard.module")  
 .then(m => m.DashboardModule),  
 canMatch: [isAuthenticated]  
 },  
 { path: '', redirectTo: '/welcome', pathMatch: 'full' },  
];  
@NgModule({  
 imports: [RouterModule.*forRoot*(appRoutes)],  
 exports: [RouterModule]  
})  
export class AppRoutingModule {}

**transaction-resource.service.ts:**

@Injectable()  
export class TransactionResourceService extends ResourceBase {  
 constructor(http: HttpClient) {  
 super(http);  
 }  
  
 public submitTransaction(target: number, amount: number): Observable<Transaction | null> {  
 return this.post(`/accounts/transactions`, {target, amount})  
 .pipe(  
 *map*((result: any) => {  
 if (result) {  
 return Transaction.*fromDto*(result);  
 }  
 return null;  
 }),  
 *catchError*((error: any) => *of*(null))  
 );  
 }  
}

**transaction.service.ts:**

@Injectable({providedIn: 'root'})  
export class TransactionService {  
 public completedTransaction: EventEmitter<Transaction | null> = new EventEmitter<Transaction | null>();  
  
 constructor(private transactionResourceService: TransactionResourceService) {}  
  
 public submitTransaction(target: number, amount: number) {  
 this.transactionResourceService.submitTransaction(target, amount).subscribe({  
 next: (v) => this.completedTransaction.emit(v),  
 error: () => this.completedTransaction.emit(null)  
 });  
 }  
}

**Transaction.component.ts:**

@Component({…})  
export class TransactiontableComponent implements OnInit {  
 public transactions: Transaction[] = [];  
  
 constructor(private transactionService: TransactionService) {}  
  
 ngOnInit(): void {  
 this.transactionService.transactionsChange  
 .subscribe((data: Transaction[]) => {  
 this.transactions = data;  
 });  
 this.getTransactions();  
 }  
  
 ngOnDestroy(): void {  
 this.transService.unsubscribe()  
 }  
  
 public getTransactions() {  
 this.transactionService.getTransaction();  
 }  
}

## asp.net

**Add.cshtml:**

@page  
@using Pizza.Models  
@model Pizza.Pages.Order.AddModel  
  
@{ ViewData["Title"] = "Add"; }  
   
<div class="row">  
 <h2>Neue Bestellung erfassen</h2>  
</div>  
<div class="row">  
 <div class="col-md-4">  
 <form method="post">  
 <div asp-validation-summary="ModelOnly" class="text-danger"></div>  
 <div class="form-group">  
 <label asp-for="Order.Name" class="control-label"></label>  
 <input asp-for="Order.Name" class="form-control" />  
 <span asp-validation-for="Order.Name" class="text-danger"></span>  
 </div>  
 <div class="form-group">  
 <input type="submit" value="Create" class="btn btn-primary" />  
 </div>  
 </form>  
 </div>  
</div>  
@section Scripts { @{await Html.RenderPartialAsync("\_ValidationScriptsPartial");} }

**Add.cshtml.cs:**

public class AddModel : PageModel {  
 private readonly ApplicationDbContext \_context;  
  
 [TempData]  
 public bool IsNew { get; set; }  
  
 [BindProperty]  
 public NewOrderViewModel Order { get; set; }  
  
 public AddModel(ApplicationDbContext context){  
 \_context = context;  
 }  
 public async Task<IActionResult> OnPostAsync(){  
 if (!ModelState.IsValid) { return Page(); }  
  
 var toAdd = new Models.Order() {Name = Order.Name};  
 \_context.Order.Add(toAdd);  
 await \_context.SaveChangesAsync();  
  
 IsNew = true;  
 return RedirectToPage("./Detail", new {toAdd.Id});  
 }  
}

**AddOrderViewModel.cs:**

public class NewOrderViewModel {  
 [Display(Name = "Pizza Name")]  
 [StringLength(20, MinimumLength = 3)]  
 [Required]  
 public string Name { get; set; }  
}