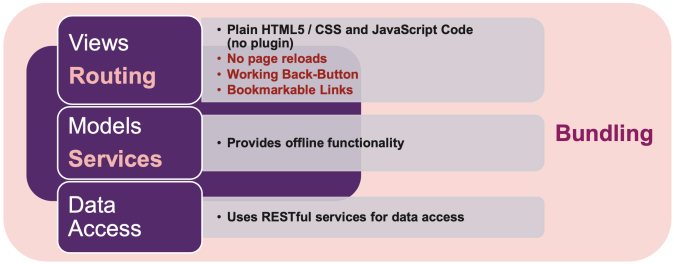
WED3 Summary

SPA-Überblick

Historisch: 1990 nur statische Seiten, ab 1995 wenig JavaScript in Seiten. 2005 Erfindung von Asynchronous JavaScript and XML, 2014 Release von HTML spezifisch for SPA. 2015 Google pusht PWA (Benachrichtungen, Service Workers, Web App Manifests). Browser werden immer mächtiger: Kamera-Zugriff, Bluetooth, Gaming Devices etc. können angesteuert werden. Browser ist ein Meta Layer (eigentliche Idee hinter Java).
Browser-basierte Applikationen funktionieren von überall her, jederzeit. Ermöglicht SaaS, keine Software-Updates nötig, können verpackt werden für Clients (Electron) oder Apps (NativeScript). Nachteile: Kein direkter Hardware-Zugriff, Applikationen tendenziell ineffizienter, komplexere Deployment-Strategien.
Traditionelle Architektur: Jeder Aufruf rendert eine neue Seite in HTML. SPA: Interaktion über Anpassung des DOMs, Server bietet APIs (mehr Logik im Client).
Charakter von SPAs: Nur HTML5 / CSS / JS, keine Page Reloads, funktionierender Zurück-Button, Lesezeichen funktionieren, limitierte Offline-Funktionalitäten.



Bundeling: Gesamter JavaScript-Code muss über tendenziell langsame Leitung zu Kunden, bundling und minifying reduziert Grösse, grosse SPAs brauchen vernünftiges Dependency Management, Module können auch On-Demand geladen werden, Bundler kommen und gehen (z.B. Webpack, Grunt, Rollup, esbuild)
Webpack:

- Entry: Startpunkt wo Webpack mit Bundling beginnt und Dependencies findet.
- Output: Wo sollen die finalen Dateien hingeschrieben werden?
- Loaders: Transformiert Dateien in Module.
- Plugins: Können zusätzliche Funktionalitäten bieten (z.B. Asset Management)
- Mode: Aktivierung bestimmter Optimierungstechniken nach Bedarf.

Routing: Wird in SPAs komplett client-seitig gemacht, Browser "fakt" URL-Änderungen, Content muss für Zurück-Button persistiert werden. Früher gelöst mittels #, heute mit `window.history` / `window.history.pushState`, verhindert das der Browser die URL wirklich lädt. Meistens gelöst über eine Routing-Tabelle, welche je nach verlangter Route eine andere Funktion aufruft.

Dependency Injection: Reduziert Kopplung zwischen Konsument und Implementation, "Verträge" zwischen Klassen basieren auf Interfaces, erlaubt flexible Ersetzung einer konkreten Implementieren.

React

Eine Bibliothek, kein Framework! Umfasst nur das V aus MVC.
Prinzip von React: Komplexe Probleme in kleinere Komponenten aufteilen. Verbessert Wiederverwendbarkeit, Erweiterbarkeit, Wartbarkeit, Testbarkeit, Aufgabenverteilung im Team.
JSX: React-Komponenten sind Funktionen, welche HTML zurückgeben können (JSX). JSX kann an beliebigen Stellen verwendet werden, wenn Dateiendung stimmt. In eckigen Klammern stehen dann JavaScript-Expresssions. Einschränkung: React-Elemente müssen mit Grossbuchstaben starten, `className` anstatt `class` verwenden wegen gesperrter Keywords. Unterelement sind mittels `props.children` zugänglich. `props` als read-only behandeln!

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

function App() {
  return (
    <Container><HelloMessage name="OST"/></↩
    Container>
  )
}
```

Styles werden als Objekt übergeben, muss Camel Case verwenden (`min-height` wird zu `minHeight`). Die JSX-Elemente werden zu `React.createElement` umgewandelt, daher muss in jedem JSX-File React importiert werden, auch wenn es nicht aktiv verwendet wird.

```
function Container(props) {
 return React.createElement("div",
  { className: "container" },
  props.children
 )
}
```

React-Kompontenten konnten früher mittels Klasse definiert werden. Seit den Hooks aber nicht mehr nötig.

```
class HelloMessage extends React.Component {
 render() {
  return <div>Hello {this.props.name}</div>
 }
}
```

Anlegen einer neuen App: `npx create-react-app hello-ost`. Konfiguration kommt dann aus einem NPM-Paket (Webpack, Babel, etc.). Kann mittels `eject` entfernt werden.

Mount: Komponenten müssen mittels Instruktion gemountet werden. Theoretisch mehrere Mounts pro Webseite möglich.

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

const root = ReactDOM.createRoot(document.↩
    getElementById('root')); root.render(<App ↩
    />);
```

State: Mittels `useState` Hook. `useState` müssen immer in derselben Reihenfolge erfolgen, somit if-Konditionen nicht möglich. State einer Komponente ist immer privat, kann aber als Props weitergegeben werden. Auch Event-Handler / Setter können als Props an Komponenten weiter gegeben werden. Zustand darf ausschliesslich mit Settern geändert werden.

```
import { useState } from 'react';

function Counter() {
 const [counter, setCounter] = useState(0);
 const increment = () => setCounter(counter + 1)↩
    ;
 return (
  <div>
   <p>{counter}</p>
   <button onClick={increment}>Increment Counter↩
    </button> </div>
 )
}
```

Reconciliation: React-Komponenten werden als virtueller DOM gerendert, Wird der State geändert, erstellt React einen neuen virtuellen DOM, alter und neuer DOM werden verglichen, erst dann werden geänderte DOM-Knoten im Browser erstellt.
Formulare: Event Handler bei den Inputs registrieren und Zustand ändern.

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

Oder mittels `onSubmit` auf dem Formular abfangen.

```
function handleSubmit(event) {
  event.preventDefault();
  alert("Username: " + username + ", Password: "↩
     + password)
}
```

Styling: Meistens mittels Widget-Library, z.B. Reactstrap, Material UI oder Semantic.
Lifecycles: Klassenkomponenten haben eine Reihe an Lifecycle-Methoden wie `componentDidMount` , `shouldComponentUpdate(nextProps, nextState)` oder `componentWillUnmount`. Mit Hooks vereinfacht mit `useEffect`. `useEffect` kann mit Promises verwendet werden.

```
useEffect(() => {
 const timerID = setInterval(() => setDate(new ↩
    Date()), 1000) // ausgefuehrt beim Mount
 return () => {
  clearInterval(timerID) // ausgefuehrt beim ↩
    Unmount
 }
}, []) // Arrays von Dependencies, kann genutzt ↩
    werden, um Effekt auszuloesen, wenn sich ↩
    Abhaengigkeit aendert
```

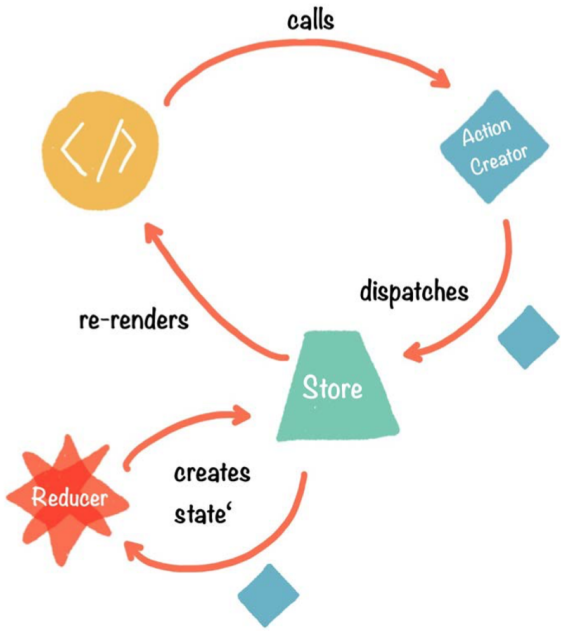
Routing: Mittels React Router (Kollektion von Navigationskomponenten für React, für Web und Native). Alle Router müssen Teil von `<BrowserRouter>` sein.
`<Route path="/about" element ={<About/>} />`: Component About wird nur gerendert, wenn der path matcht. App-interne Links verwenden nicht ¡a¿ sondern ¡Link¿.
`<Link to="/about">About</Link>`

Type-Checking: Flow erweitert JavaScript um Typannotationen. Lieber Typescript für mehr Typsicherheit in React-Komponenten. Flow sind nur Annotations, können daher einfach ignoriert werden, Typescript ist eine ganze Programmiersprache.

React Context: Daten immer als Props mitgeben ist mühsam, Zustand verteilt sich über gesamte Applikation, Calls sind auch verteilt. React Context ermöglicht es, Props für alle Unterkomponenten zur Verfügung zu stellen.

```
const ThemeContext = React.createContext(themes.↩
    light);

function App() {
 return (
  <ThemeContext.Provider
   value={themes.dark}>
   <Toolbar />
  </ThemeContext.Provider>
 );
}

function ThemedButton() {
 const theme = useContext(ThemeContext);
 return (
  <button style={{
   background: theme.background,
   color: theme.foreground
  }}>
   {" "}I am styled by theme context!{" "}
  </button>
 );
}
```

Redux: Darstellung des States als Baum, Baum ist nicht veränderbar, Veränderungen am Baum führen zu einem neuen Baum, Verwaltung über Stores.



Eine Veränderung braucht eine Action (sehr simple Objekte wie `{ type: 'TRANSFER', amount: 100 }`). Der Store braucht einen Reducer, um mit der Action den neuen Baum zu machen. Reducer sind pure Funktionen ohne Seiteneffekte.
Soll / Muss in jeder React-Applikation Redux eingesetzt werden? Nein, wenn kaum Zustand existiert, der von mehreren Komponenten verwendet wird, lohnt sich der Redux-Overhead nicht.

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

Mehrere Reducer bilden einen Root Reducer. Initialer State für die App ist ein leeres Objekt.

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

// gleichwertig
const rootReducer = combineReducers({
  balance,
  transactions
})

const store = createStore(rootReducer);
```

Über Änderungen am State kann man sich mittels Listener benachrichtigen lassen:
`store.subscribe(()=> console.log(store.getState()));`
React und Redux: Redux Toolkit verwenden. `createSlice` erstellt neue Stateobjekte, Reduce-Funktionen und Aktionen. Action-Type im unteren Beispiel ist `balance/transfer`. Mittels immer.js scheinbare, direkte Änderungen am State möglich.

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

`configureStore` initialisiert den Redux Store mit den angegeben Reducern. Enthält `redux-thunk`. Redux Thunk erlaubt es uns, anstelle eines Objektes eine Funktion zu dispatchen.

```
const store = configureStore({
 reducer: { balance: balanceReducer }
});
```

Verfügbarkeit in React-Applikation mittels Provider.

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

`useDispatch` wird für den Dispatch der Aktionen an den Store benutzt. `useSelector` wird für die Abfrage des States benutzt.

```
const dispatch = useDispatch()
dispatch(transfer({ amount: 10 }))
const balance = useSelector(state => state.↩
    balance.value);
```

Asynchrone Actions:

```
// First, create the thunk
export const transferAsync = createAsyncThunk(
 "balance/transferApiRequest",
 async (amount) => {
```

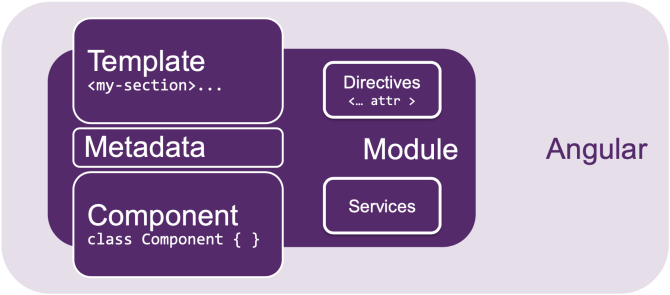
```
  const response = await api.transfer(amount);
  return response.data;
 }
);

const balanceSlice = createSlice({
 initialState: { value: 0, status: "idle" },
 extraReducers: (builder) => {
  builder
   .addCase(transferAsync.pending, (state) => {
    state.status = "loading";
   })
   .addCase(transferAsync.fulfilled, (state, ↩
    action) => {
    state.status = "idle";
    state.value += action.payload.amount;
   });
 },
});
```

JHipster: Fullstack App-Generator mit Angular, React-Redux oder Vue, Spring Boot, Maven/Gradle, NPM, Postgres, MongoDB, Elasticsearch, Cassandra, Kafka etc. Bieten eigene DSL für Entities und Relationen.

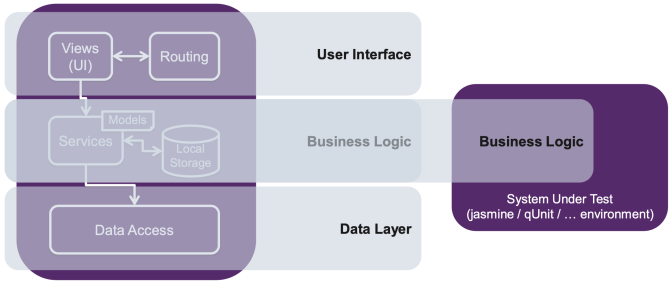
Testing: Jest offizielle Lösung von Facebook, kommt mit `create-react-app` mit. Die React Testing Library baut auf der DOM Testing Library auf und fügt APIs für die Arbeit mit React-Komponenten hinzu. JHipster generiert End-to-End-Tests mit Cypress.

```
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



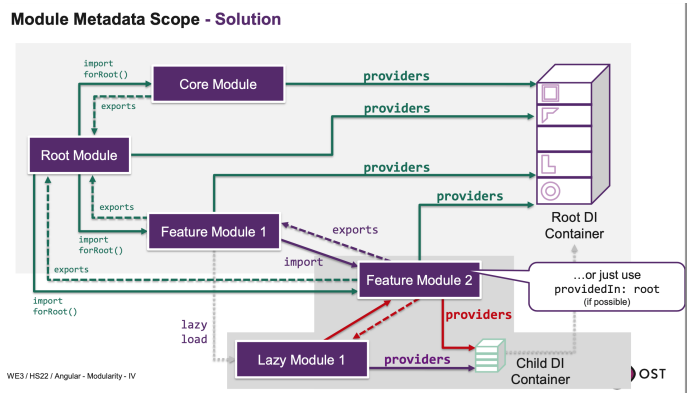
TypeScript-basiert, mit Dependency Injection, 2-Way-Bindings, klar strukturiert. Sollte verwendet werden für lang lebende und komplexe Applikationen. Historie: Modernes Angular seit v2, v1 wird AngularJS genannt, keine Gemeinsamkeiten. v3 ausgelassen, um Paketnamen zu harmonisieren. `npx ng new my-app` legt neue Applikation an (Paket lokal installieren mit `npm install @angular/cli`)



Dependency Injection: Registration beim Container, Request, Resolve durch Container, Fullfill (TypeScript module/s).
ngModules: Ein zusammenhängender Codeblock, der eng miteinander verbundenen Fähigkeiten gewidmet ist (TypeScript class). Jede App hat mindestens ein Modul, das Root-Modul. Exportieren Features wie Services oder Direktive für andere Module. ngModule-Deklaration selbst wird in ein TypeScript-Modul eingefügt (meistens über `index.ts`).

```
@NgModule({
  exports: [] // The subset of declarations that↩
     should be visible and usable in the ↩
    component templates of other modules.
  imports: [CommonModule], // Specifies the ↩
    modules which exports/providers should be ↩
    imported into this module.
  declarations: [], // The view classes that ↩
    belong to this module (components, ↩
    directives and pipes).
  providers: [], // Creators of services that ↩
    this module contributes to the global ↩
    collection of services (Dependency Injection↩
     Container); they become accessible in all ↩
    parts of the app.
  bootstrap: [] // The main application view, ↩
    called the root component. Only the root ↩
    module should set this property.
})

export class CoreModule { }
```

Module-Importe: Standardmässig wird alles aus dem Modul importiert und Dependency-injected. `forChild(config?)`: Statische Methode auf dem Modul, erlaubt Services für den aktuellen Modul-Level zu konfigurieren. `forRoot(config?)`: Statische Methode auf dem Modul, injected und konfiguriert Services global. Nur im Root-Modul machen! `providedIn: 'root'` bevorzugen, wenn Services keine Konfiguration benötigen.



Modultypen: Root / App Modul: Einstiegspunkt in die Applikation. Exportiert nichts. Konventionell AppModule genannt und existiert in einer Datei namens app.module.ts. Importiert BrowserModule, die jede Browser-Anwendung importieren muss
Feature-Modul: Teile der Applikation. Es ist die beste Praxis, Feature-Module in Domain-, Routing-, Service-, Widget- und Lazy-Module aufzuteilen (siehe unten). Ermöglicht die Zuweisung von Entwicklungsaufgaben an verschiedene Teams.
Shared-Modul: "Toolkit" der Applikation, alles was irgendwie in kein Modul passt. Keine app-weiten Singleton-Anbieter (Dienste) in einem gemeinsamen Modul angeben.
Core-Modul: Initialisiert globale Services. Ein "lazy-loaded"-Modul, das dieses gemeinsame Modul importiert, erstellt seine eigene Kopie des Dienstes. Wird nur vom Root-Modul importiert, der Import in ein anderes Modul, insbesondere in ein "Lazy-Load"-Modul, widerspricht der Absicht und kann zu einem Laufzeitfehler führen.
Lazy Modules: Ähnlich wie Feature-Modules, werden aber lazy loadet wenn angefragt mittels Dependency Injection.
Feature-Modul-Untertypen: Domain-Modul: Bereitstellung einer auf einen bestimmten Anwendungsbereich zugeschnittenen Benutzeroberfläche. Routing-Modul: Gibt die Routing-spezifischen Konfigurationseinstellungen des Feature- (oder Root-) Moduls an. Service-Modul: Bietet Versorgungsdienste wie Datenzugriff und Nachrichtenübermittlung. Widget Modul: Macht Komponenten, Direktiven und Pipes für externe Module verfügbar.

Directives: Enthält Anweisungen zur Transformation des DOM (TypeScript class). Besitzen kein Template. Brauchen `@Directive()`-Decorator. Structural directives: Verändern DOM. Diese werden im Hintergrund zu `<ng-template>` umgewandelt. `<ng-template>` kann auch verwendet werden, wenn kein HTML-Element benötigt wird. `<ng-template>` können nicht mit weiteren Structural directives verwendet werden. Attribute directives: Ändern des Aussehens oder Verhaltens eines vorhandenen Elements.

```
<div *ngIf="hasTitle"><!-- shown if title ↩
    available --></div>
<div [ngStyle]="{ 'font-size': isSpecial ? 'x-↩
    large' : 'smaller' }">
     <!-- render element -->
</div>
```

Template Reference Variables: Verfügbar im gesamten Template.

```
<input placeholder="phone number" #phone>
<button (click)="callPhone(phone.value)">Call</↩
    button>
```

Components: Eine Komponente ist eine Richtlinie mit einer Vorlage; sie steuert einen Abschnitt der Ansicht (HTML File / (S)CSS / ...). Basiert auf MVC oder MVVM. Eine Komponente sollte so klein und zusammenhängend wie möglich implementiert werden, um die Testbarkeit / Wartbarkeit / Wiederverwendbarkeit zu unterstützen. Komponenten kontrollieren die View (genau eine View pro Komponente). Mittels Selektor kann Komponent in anderen Views verwendet werden (entweder tag-name oder CSS-Selektor (id-selector #topHeader)), braucht Registierung im ngModule bei `declarations` und `exports`. Lifecycle wird verwaltet von Angular (Hydration, Update, Dehydration), können mittels Methoden wie `ngOnInit` und `ngOnDestroy` erweitert werden. Komponent braucht Deklaration wie `implements OnInit, OnDestroy`, damit Methoden verwendet werden können. View-Code muss gültiges HTML5 sein.

```
<p>Your team is <strong>{{counter.team}}</strong↩
    ></p>
<p>Your current count is
  <strong>{{counter.count}}</strong>
</p>
<form>
  <button (click)="up($event)">Count Up</button>
</form>

@Component(...)
export class CounterComponent implements OnInit ↩
    {
  counter: CounterModel = new CounterModel(); up↩
    (event: UIEvent): void {
    this.counter.count++;
    event.preventDefault();
  }
}
```

Bindings: Two Way Binding / Banana in a box [(...)]. One Way (from View to Model / Event Binding) (...). One Way (from Model to View / Property Binding) [...] or ... .

```
public counter: any = {
  get team() { return null }, set team(val) { },↩
     eventHandler: () => { }
}

<input type="text" [(ngModel)]="counter.team">
<button (click)="counter.eventHandler($event)">
<p>... {{counter.team}} ..</p>
```

Die Bindung an Ziele muss als Inputs oder Outputs deklariert werden.

```
@Component({...})
export class NavigationComponent {
  @Output() click = new EventEmitter<any>();
  @Input() title: string;
}

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

Metadata: Metadaten beschreiben eine Klasse und sagen Angular, wie sie zu verarbeiten ist (TypeScript decorator).
Services: Bietet Logik für jeden Wert, jede Funktion oder jedes Merkmal, das Ihre Anwendung benötigt (TypeScript class). Werden mittels Dependency Injection erstellt, wenn Komponenten Service-Abhängigkeit deklarieren. **Service-Kommunikation mit UI**: Theoretisch alles mittels RxJS möglich, in WED3 aber mit `EventEmitter` behandelt (siehe oben). RxJS implementiert das Observer-Pattern für JavaScript. **Hot Observables**: Sequenz von Events wie Mausklicks, verfügbar für alle Subscriber. **Cold Observables**: Starten erst bei ersten Subscriber, schliessen sobald Task beendet. **Data Resources**: Abstrahieren HTTP-Kommunikation, um Daten abzuholen. Jeweils `HttpClient` verwenden, verwendet Cold Observerables.

```
export class SampleModel { }

@Injectable({ providedIn: 'root')
export class SampleService {
  private samples: SampleModel[] = []; // simple↩
     cache
  public sampleChanged: EventEmitter<SampleModel↩
    []> = new EventEmitter<SampleModel[]>();
  constructor(
    private dataResource: ↩
    SampleDataResourceService) {
  }
  load(): void {
    this.dataResource.get().subscribe(
      (samples: SampleModel[]) => { // update ↩
    cache, emit change event, ...
        this.samples = samples;
        this.sampleChanged.emit(this.samples);
      });
  }
}

@Component({ ... })
export class SampleComponent implements OnInit, ↩
    OnDestroy {
  private samples: SampleModel[];
  private samplesSubscription: Subscription;
  constructor(private sampleService: ↩
    SampleService) { }

  ngOnInit() {
    this.samplesSubscription = this.↩
    sampleService.samplesChanged.subscribe(
      (data: SampleModel[]) => { this.samples = ↩
    data; });
  }

  ngOnDestroy() {
    this.samplesSubscription.unsubscribe();
  }
}
```

HTTP-Requests können abgefangen und modifiziert werden.

```
  @NgModule({ ...})
  export class SampleModule {
    static forRoot(config?: {}): ↩
    ModuleWithProviders {
      return {
        ngModule: SampleModule,
        providers: [
          {
            provide: HTTP_INTERCEPTOR,
            useClass: AuthInterceptor,
            multi: true
          }
        ]
      }
    }

    @Injectable()
    export class AuthInterceptor implements ↩
      HttpInterceptor {
      constructor(private store: ↩
      SecurityTokenStore) { }

      public intercept(req: HttpRequest<any>, next↩
      : HttpHandler) {
        const authReq = req.clone({
          setHeaders: {
            Authorization: `Bearer ${this.store.↩
      token}`
          },
```

```
        withCredentials: true
      });

      return next.handle(authReq);
    }
  }
```

Template-driven forms: Angular Template-Syntax mit den formularspezifischen Direktiven und Techniken. Erzeugt weniger Code, platziert aber die Validierungslogik in HTML. **Reactive (or model-driven) forms**: Import von `ReactiveFormsModule` nötig, Form und Validations werden im Controller gebaut, mehrere asynchrone Validierungen möglich. Nicht Teil der Lektüre!

```
<form (ngSubmit)="doLogin(sampleForm)" #sampleForm="ngForm">
  <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
  </div>

  <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
      Component
      return false; // avoid postback }
    }
  }
}
```

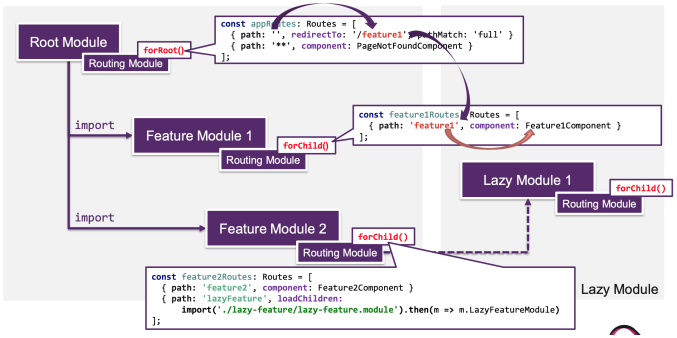
Component Transclusion / Content Projection: Selbiges wie Reacts `props.children`.

```
<wed-navigation>
  <h1 wed-title>WED3 Lecture</h1>
  <menu><!-- ... --></menu>
</wed-navigation>

<header>
  <ng-content select='[wed-title]'> </ng-content>
</header>
<nav>
  <ng-content select='menu'>
  </ng-content>
</nav>

<wed-navigation>
  <header>
    <h1 wed-title>WED3 Lecture</h1>
  </header>
  <nav>
    <menu><!-- ... --></menu>
  </nav>
</wed-navigation>
```

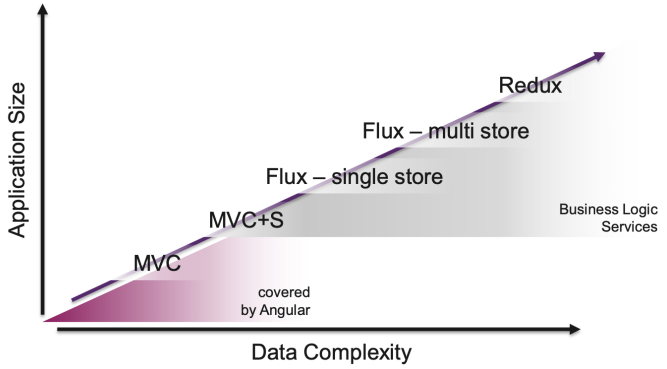
```
var subscription = this.http.get('api/samples').
    subscribe(
  function (x) { /* onNext -> data received (in x) */ },
  function (e) { /* onError -> the error (e) has been thrown */ },
  function () { /* onCompleted -> the stream is closing down */ }
);
```

Routing: Routing Table mappt Routen zu Kompontenten. Oberste Routen registrieren mit `forRoot()`, Subroutes mit `forChild()`. `forRoot()` kennt auch Router-Service, welcher Singleton ist in Applikation. Benötigt `<base href="/">` im Header für relative URL oder `APP_BASE_HREF` Variable. `<router-outlet>` definiert, wo Routen angezeigt werden. `'hero/:id'` ID ist hier eine Variable. `redirectTo` erlaubt Weiterleitung. `'**'` ist die Wild-Card-Route für 404. Routen werden aufgelöst nach "first come first served". Verschachtelung möglich mittels `children`, Parent benötigt dann weiteres Router Outlet.

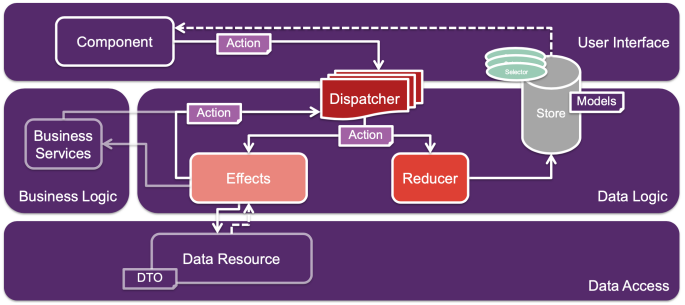
```
const appRoutes: Routes = [
 {
   path: '', component: WelcomeComponent,
   children: [
     { path: ':id', component: SamplesDetailComponent },
   ]
 },
 {
   path: 'config',
   loadChildren: () => import('./cfg/cfg.module').then(m => m.CfgModule), canLoad: [AuthGuard]
 }
];

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



Angular-Archictekturen: Flow architekturmässig dasselbe wie Redux. Redux-Anbindung mittels `ngxs` oder `ngrx`.



Vorteile ngrx: Einfaches Debugging, Undo/Redo einfache Implementation, weniger Code in Komponenten, weniger Change Detection Overhead. Nachteile: Komplexere Architektur, Datenlogik fragmentiert in Reducer / Effects, zusätzliche Bibliothek.



MVC+S: Daten-basierte Logik gekapselt in Services mittels RxJS. `BehaviorSubject` sendet ersten State bei Aufruf (normales Subject erst beim ersten Change, UI würde leer bleiben). Sollte nicht public gemacht werden, ansonsten können Clients `next()` aufrufen.

```
@Injectable({ providedIn: 'root' })
export class SampleService {
  private samples: BehaviorSubject<SampleModel[]> = new BehaviorSubject([]); // Event bus
```

```
    which is used to store the last state and to↩
     notify subscribers about updates.
  public samples$: Observable<SampleModel[]> = ↩
    this.samples.asObservable(); // Convert ↩
    event bus into an observable, which can be ↩
    provided to the UI or other services.

  constructor(
    private resourceService: ↩
    SampleResourceService { }
  )

  public addSample(newSample: SampleModel): ↩
    Observable<any> {
    return this.resourceService
      .post(newSample)
      .pipe(
        tap(res => {
          this.samples.next([...this.samples.↩
    getValue(), newSample]); // Store the ↩
    retrieved data into the BehaviorSubject and ↩
    emit the data changed event. Create a new ↩
    array, otherwise the pipe cannot track the ↩
    change.
        }),
        catchError((err) => this.handleError(err↩
    )));
  }
}
```

Pipe: Hilfreich für kleine Transformationen, erlauben Parameter (`<p>{{counter.date | date:'longDate'}}</p>`). Pure Pipes werden nur ausgeführt bei Änderungen am Subjekt, impure laufen bei jeder Change Detection. Vorgefertige Pipes: `async`, `number`, `date`, `percent`. Async erlaubt direkte Anzeige ab Observable (`<li *ngFor="let s of sampleService.samples$ | async">`) Filtering und Ordering Pipes gibt es nicht, gelten als Impure, sollte im Komponent gemacht werden.

```
@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;
  }
}
```

Styling: Styling standardmässig nur auf aktuellen Komponent `Emulate`, kann mit `encapsulation` kontrolliert werden, `Native` verwendet Shadow DOM von Browser, `None` fügt alle Styles global zusammen. Spezial-Sachen: `:host`: Zeigt auf das aktuelle Element. `host-context`: Sucht Vorgänger ab (`:host-context(.theme-light)h2 { }`) Drei Wege für Styling: `styles`-Array auf Komponent, `styleUrls`-Array auf Komponent, zeigt auf (S)CSS-Dateien, `template` auf Komponent direkt.
Angular-Tooling: Angular State Inspector, Redux DevTools, Ahead-Of-Time (AoT) Compilation.

.NET

Attribute: intensiv eingesetzt (Um Konventionen zu übersteuern bzw zu unterstützen, Schnittstelle für Zusatzinformationen für Framework, Kann via "Reflection" ausgelesen werden).

```
[Required]
[StringLength(100, MinimumLength = 10)]
public string Name {get; set; }

[HttpPost]
public ActionResult About()
```

Async Await

```
static async Task Main(string[] args)
{
 await RunAsync();
}
public static Task<string> Send()
{
 return Task.Run(() => {
  Console.WriteLine("Send!");
  return "nachricht gesendet";
 });
}
public static async Task<bool> RunAsync()
{
 Console.WriteLine("Start Send");
 Console.WriteLine(await Send());
 Console.WriteLine("End Send");
 return true;
}
```

MVVM View: Markup Language, was Benutzer sieht, Kommunikation zu ViewModel via Bindings. View Model: Daten für View aufbereiten, Value Converter, UI Logik. Model: Daten, Services, Domain Logik Dependency Injection - Lifetimes: Transient: created each time they are requested. Best for lightweeight, stateless services. Scoped: created once per request. Singelton: created the first time they are requested and then every subsequent request will use the same instance. **Pages**: Routing: generiert anhand von URL Antwort, URL wird auf Aktion: "gemappt" (Routing-Module). Bei Aufruf im Folder "/pages/" nach Page gesucht und ausgeführt (default case insensitive) MVVM: Besteht aus 2 Files, *.cshtml (View mit Razor), *.cshtml.cs (Viewmodel)

```
@page
@model HelloWorldModel

@{
 ViewData["Title"] = "HelloWorld";
}

<h1>@Model.HelloWorld</h1>

public class HelloWorldModel : PageModel
{
 public string HelloWorld {get;set;}
 public void OnGet()
 {
  HelloWorld = "Hi World!";
 }
}
```

@model: Beschreibt Type vom View Model dieser Seite, wird in Var Model abgelegt PageModel: Basis Klassse mit Hilfsfunktionen (Redirects / Event-Hooks), Aufbereitung der Daten für die View MVVM - Model: kann pro HTTP-Verb eine Funktion definieren die davor aufgerufen wird (OnGet, OnPost), Body und Query werden automatisch gemappt, Parameter werden als Argumente übergeben, können auch alss Klasse entgegen genommen werden. Mit [BindProperty] kann auf das Kopieren von Properties verzichtet werden, nutzen falls Properties 1:1 der View übergegebn werden [BindProperty(SupportsGet = true)]

```
public class PostModel : PageModel
{
 [BindProperty]
 public string EchoText {get; set;}
}
```

View: @page definiert Razor-File als Page, @page "/test/id?" überschreibt Default-Routing Informationen kann auf verschiedene Arten zugegriffen werden:

```
@page "/test/{id:int?}"
@model Ex.Pages.Page.RoutingModel
@{
 ViewData["Title"] = "Routing";
}
<h1>Routing</h1>
@RouteData.Values["id"]

public class RoutignModel : PageModel
{
 public int Id {get;set;}
 [BindProperty(SupportsGet = true, Name="Id")]

 public int Id2 {get;set;}
 public void OnGet(int id)
 {
  Id2=id
 }
}
```

Tag Helpers: Ermöglichen Code an HTML Tags zu binden

```
<email mail-for="support@examl.com"></email>
<a href="mailto:support@example.com">↩
   support@example.com</a>

public classs EmailTagHelper : TagHelper
{
 public string MailFor {get;set;}
 public override void Process(TagHelperContext ↩
   context, TagHelperOutput ouput)
 {
  output.TagName = "a";
  output.Attributes.SetAttrbutes("href", "↩
   mailto:" + MailFor);
  output.Content.SetContent(MailFor);
 }
}
```

AJAX Handlers: Pages können weiter Actions als "handler" anbieten; Schema; On[Method][Name] z.B. OnPostEcho, aufgerufen via: [METHOD] /[PAGE]?handler=[HandlerName] -¿ GET /Ajax?handler=autocomplete

Entity Framework (EF): objektrelationale Zuordung, die Entwickler über domänenspezifsche Objekte die Nutzung relationaler Daten ermöglicht. (wenig Code, viele Konvention, OR-Mapper). Code First benötigt; type discovery: welche Klassen gehören in die DB, Connection String: Wohin mit Daten, DbContext: Entry Point

```
public class Order
{
 public long Id {get;set;}
 [Required]
 public string Name {get;set;}
 public DateTime Date {get;set;}
 public virtual ApplicationUser Customer {get;↩
   set;}
 public Order() {
  Date = DateTime.Now;
 }
}
```

public [long/string] Id -¿ automatisch Primary key von Entity public virtual ApplicationUser Customer -¿ automatisch als Navigation Properety erkannt (CustomerId -¿ wird als Foreign Key für Customer Property erkannt) [Required] -¿ NotNull, [NotMapped] -¿ nicht in DB geschrieben, [Key] -¿ Definiert Primary Key der Entity, [MaxLength(10)] -¿ Beeinflusst die Allokatingrösse des Feldes in DB

Validation: Client: JQuery Valdation, Server: ASPNET, z.B. Annotieren der Klasse z.B. [Required] oder [StringLength(60, MinimumLength=3)] (Sind kombinierbar), Razor anpassen (Validation ins DOM einfügen), Server Side validierung

ASPNET identity features: Autorisierung: via Attrbute [Authorize(Roles="Admin, PowerUser")] oder [Authorize(Policy="OlderThan18")] oder via Services z.B. await _userManager.isInRoleAsync(user, "Admin")

API Routing: Funktioniert über Attribute [Route] definieret einen neuen Eintag im Router, [HttpMethod] bei Actions ist required [Route("api/[controller]")] -¿ Klassen def -¿ [HttpGet] public IEnumerable¡Value¿ Get() oder [HttpGet("id")] public Value Get(int id)