1 Introduction

1.1 Browser-based Applications

Benefits:

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

Liabilities:

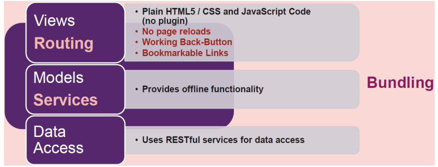
- No data sovereignty (Datenhoheit)
- Limited/restricted hardware access
- SEO - Search engines must execute JavaScript
- More complex deployment strategies

1.2 SPA

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

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

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


Views
Routing
• Plain HTML5 / CSS and JavaScript Code (no plugin)
• No page reloads
• Working Back-Button
• Bookmarkable Links
Models
Services
• Provides offline functionality
Data Access
• Uses RESTful services for data access
Bundling

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

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

Bundling SPAs: E.g. WebPack

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

Dependency Injection Benefits:

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

2 React

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

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

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

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

2.1 JavaScript XML

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

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

2.1.1 Conditionals

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

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

oder

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

2.1.2 Funktions- und Klassenkomponenten

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

2.1.3 Props

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

2.1.4 Rendering und Mounting

Mounting: Nötig um Komponenten auf Webseite anzuzeigen.

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

2.2 React State

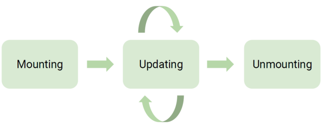
React-Klassenkomponenten können einen veränderbaren Zustand haben. Der **state** einer Komponente ist immer privat. Ändert der State, wird auch die Komponente aktualisiert.

```
class Counter extends React.Component {
  state = { counter: 0 }
  increment = () => { this.setState({
    counter: this.state.counter + 1 }) }
  render() { return (<div> {this.state.counter}
    <button onclick={this.increment}>Increment</button>
  </div>) } }
```

2.3 Formulare mit React

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

2.4 Komponenten Lifecycle



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

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

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

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

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

2.5 React Router

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

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

2.6 Hooks

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

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

State Hook:

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

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

Effect Hook:

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

2.7 Typechecking

Flow:

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

TypeScript:

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

Vorteil gegenüber Flow:

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

2.8 Redux

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

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

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

Redux Reducer-Funktionen:
Reducer sind pure Funktionen, haben also keine Seiteneffekte.

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

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

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

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

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

2.9 Redux mit React verbinden

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

```
const mapStateToProps = (state) => { return {
  transactions: state.transactions } }
const mapDispatchToProps = { fetchTransactions }
export default connect(mapStateToProps,
     mapDispatchToProps)(Component);
// Root Komponente
const store = createStore(rootReducer, applyMiddleware(
     thunkMiddleware));
render( <Provider store={store}> <App /></Provider>
     document.getElementById('root') )
```

2.9.1 Thunk Actions

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

3 Angular

Flexible SPA Framework for CRUD applications

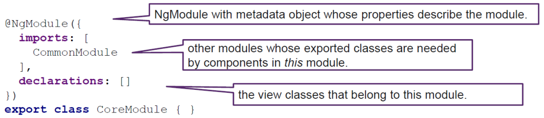
- Typescript 4.1 based
- Reduces boilerplate Code
- Dependency Injection Mechanism
- JS-optimized 2-way binding
- Clearly structured, information hiding
- Increases testability / maintainability of client-side code

ngModules: Cohesive block of code dedicated to closely related set of capabilities. (*module*) **Directives:** Provides instructions to transform the DOM. (*class*) **Components:** Directive-with-a-template; it controls a section of the view. (*class*) **Templates:** Form of HTML defining how to render the component. (*HTML/CSS*) **Metadata:** Describes a class and defines how to process it. (*decorator*) **Services:** Provides logic of any value, function or feature that the app needs. (*class*)

3.1 Angular Modules (ngModule)

Base for Angular modularity system. Every app has at least one Module, the root Module (a.k.a app). Root Module ist launched to bootstrap the app. Modules export features (directives, services) required by other modules.

TypeScript Module vs. ngModule:
ngModule is a logical block of multipe TypeScript modules linked together. The ngModule declaration itself is placed into a TypeScript module. Modules can accommodate sub-modules. All public TS members are exported as an overall *barrel*



```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class CoreModule { }
```

NgModule with metadata object whose properties describe the module.
other modules whose exported classes are needed by components in *this* module.
the view classes that belong to this module.

declarations: View Classes that belong to this module (Components, Directives, Pipes). **exports:** Subset of declarations that should be visible and usable by other modules. **imports:** Specifies the modules which exports/providers should be imported. **providers:** Creators of services that this module contributes to the global collection of services (DI 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.

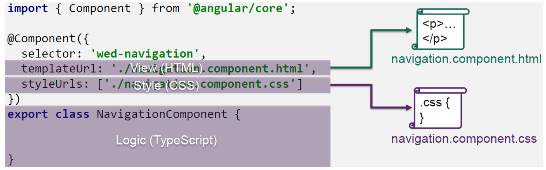
3.2 Components

A Component manages the view and binds data from the model. It consists of:

- Controller → provides logic of view, declared as TS-Class with an @component() function decorator
- HTML file → declares visual interface (template expression)
- (S)CSS file → styles behind HTML file

Components can be nested (results in a Component tree). Provide intofrmation hiding:

- Each Component declares a part of the UI
- A Component should be implemented as small coherent piece to support Testability, Maintainability, Reusability



```
import { Component } from '@angular/core';

@Component({
  selector: 'wed-navigation',
  templateUrl: './navigation.component.html',
  styleUrls: ['./navigation.component.css']
})
export class NavigationComponent {

}
```

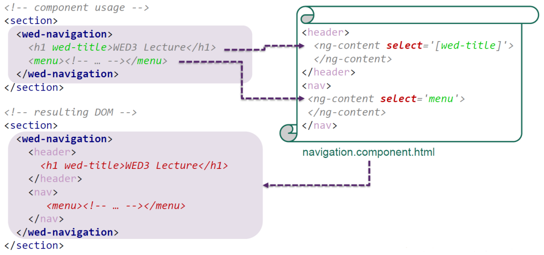
Components must be declared within the containing module so its **selector** is registered for all sub-components of that module. They can be exported, so other modules can import them.

3.2.1 Components Lifecycle

Most important events are create (ngOnInit) and destroy (ngOnDestroy). ngAfter...events are mainly for control.

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

3.2.2 Content Projection



```
<!-- component usage -->
<section>
  <wed-navigation>
    <h1 wed-title>WED3 Lecture</h1>
    <menu><!-- … --></menu>
  </wed-navigation>
</section>

<!-- resulting DOM -->
<section>
  <wed-navigation>
    <header>
      <h1 wed-title>WED3 Lecture</h1>
    </header>
    <nav>
      <menu><!-- … --></menu>
    </nav>
  </wed-navigation>
</section>
```

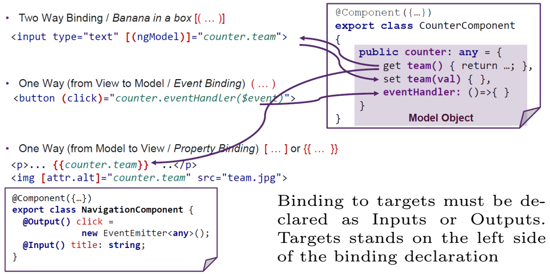
```
<header>
  <ng-content select='[wed-title]'>
  </ng-content>
</header>
<nav>
 <ng-content select='menu'>
 </ng-content>
</nav>
```

navigation.component.html

3.3 Templates

Angular extends the HTML vocabulary of your templates with: Interpolation ({{...}}), Template Expression/Statements, Binding Syntax, Directives, Template Reference Variables, Template Expression Operators

Binding:



- Two Way Binding / *Banana in a box* [(…)]

```
<input type="text" [(ngModel)]="counter.team">
```

- One Way (from View to Model / *Event Binding*) (…)

```
<button (click)="counter.eventHandler($event)">
```

- One Way (from Model to View / *Property Binding*) […] or {{ … }}

```
<p>... {{counter.team}} ..</p>
<img [attr.alt]="counter.team" src="team.jpg">
```

```
@Component({…})
export class CounterComponent
{
  public counter: any = {
    get team() { return …; },
    set team(val) { },
    eventHandler: ()=>{ }
  }
}
```

Model Object

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

Binding to targets must be declared as Inputs or Outputs. Targets stands on the left side of the binding declaration

e.g. the click / title property: `<wed-navigation (click)="..." [title]="..."> </wed-...>`

3.4 Directives

Similar to a component, but without a template. TypeScript class with an *@Directive()* function decorator.

Attribute Directives:

NgStyle **Directive**

Sets the inline styles dynamically, based on the state of the component.

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

NgClass **Directive**

Bind to the ngClass directive to add or remove several classes simultaneously.

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

Structural Directives:

NgIf **Directive**

Takes a boolean value and makes an entire chunk of the DOM appear or disappear.

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

NgFor **Directive**

Represents a way to present a list of items.

```
<li *ngFor="let element of elements"><!-- render element --></li>
```

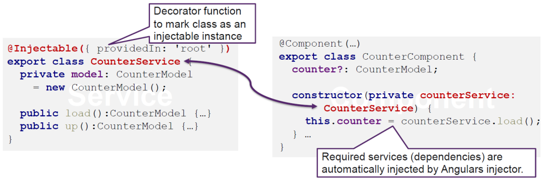
Template Reference Variables:

```
<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>
```

3.5 Services

Provides any value, function, or feature that your application needs. Typical services are: logging service, data service, message bus, tax calculator, application configuration

```
@Injectable({ providedIn: 'root' })
export class CounterService { ... }
```



Decorator function to mark class as an injectable instance

```
@Injectable({ providedIn: 'root' })
export class CounterService {
  private model: CounterModel
    = new CounterModel();

  public load():CounterModel {…}
  public up():CounterModel {…}
}
```

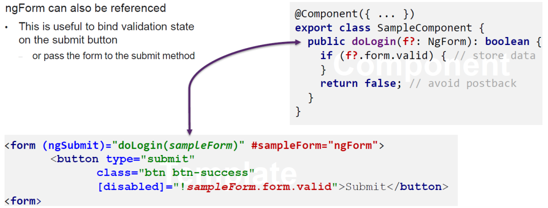
```
@Component(…)
export class CounterComponent {
  counter?: CounterModel;

  constructor(private counterService:
       CounterService) {
    this.counter = counterService.load();
  } …
}
```

Required services (dependencies) are automatically injected by Angulars injector.

3.6 Forms

```
<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>
```



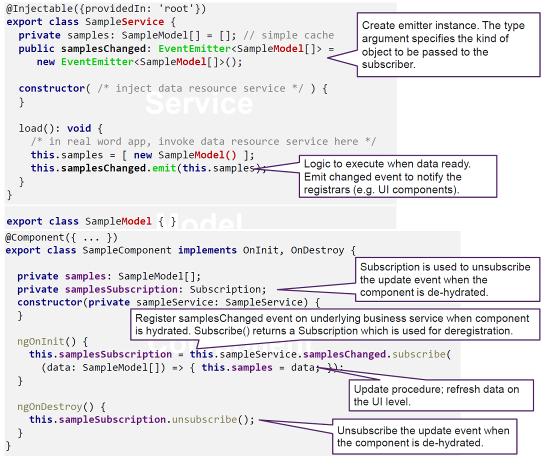
ngForm can also be referenced

- This is useful to bind validation state on the submit button
  - or pass the form to the submit method

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

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

3.7 Asynchronous Services



```
@Injectable({providedIn: 'root'})
export class SampleService {
  private samples: SampleModel[] = []; // simple cache
  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() ];
    this.samplesChanged.emit(this.samples);
  }
}

export class SampleModel { }
```

Create emitter instance. The type argument specifies the kind of object to be passed to the subscriber.

Logic to execute when data ready. Emit changed event to notify the registrars (e.g. UI components).

```
@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.sampleSubscription.unsubscribe();
  }
}
```

Subscription is used to unsubscribe the update event when the component is de-hydrated.

Register samplesChanged event on underlying business service when component is hydrated. Subscribe() returns a Subscription which is used for deregistration.

Update procedure; refresh data on the UI level.

Unsubscribe the update event when the component is de-hydrated.

3.8 Data Access

HTTP Client implements asynchronisms by using the RxJS library (library that implements the Observable pattern). **Hot Observables:** Sequence of events (mouse move), shared among all subscribers **Cold Observables:** start running on subscription (async web requests), Not shared among subscribers, automatically closed after task is finished

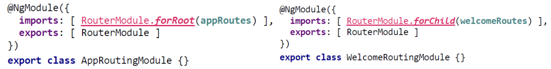
```
var subscription = this.http.get('api/samples').subscribe(
  function (x) { /* onNext -> data received (in x) */ },
  function (e) { /* onError -> the error (e) has been thrown */ },
  function () { /* onCompleted -> the stream is closing down */ }
);
```

3.9 Routing

To navigate among the views. Multiple directives: RouterOutlet, RouterLink, RouterLinkActive

.forRoot(): use exactly once to declare routes on root level

.forChild(): use when declaring sub-routings



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

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

```
const appRoutes: Routes = [
  // matches /hero/42, 42 saved in param
  {path: 'hero/:id', component: 'Hero'},
  // redirect
  {path: '', redirectTo: '/heroes', pathMatch: 'full'},
  {path: '**', component: PageNotFound} ]; // Wildcard
//Lazy Loading
{path: 'config', loadChildren: () => import('./cfg/cfg
.module').then(m => m.CfgModule), canLoad: [AuthGuard] }
```

3.10 Redux Architecture

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

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

Liabilities:

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

4 PWA & Angular & Firebase

5 ASP.NET

5.1 Test