

Komponenten und Direktiven Ökosystem

- Jede Komponente gehört zu einer ViewRef
 - Block von Elementen => Benutzeroberfläche
 - Basis für die Änderungserkennung.
- Komponenten und Direktiven DI-Eigenschaften
 - **ElementRef**
 - Element-Referenz (Zugang zum nativenElement)

Komponenten und Direktiven Ökosystem

- Komponenten und Direktiven DI-Eigenschaften
- **ViewContainerRef**
 - Referenz zum Container für dynamische Inhalte, welche hinter dem Element angehängt werden `<router-outlet>`
 - Embedded view – über Vorlagen-Referenzen
 - Host View – über Komponenten-Referenzen

Eltern Kind Beziehung

- Wiederholung der wichtigsten Dekoratoren
 - @ContentChild & @ContentChildren
 - @ViewChild & @ViewChildren
 - @Input & @Output

Inhalte transkludieren

- Komponenten stellen eine View dar.
- Beschrieben wird die View in der HTML-Vorlage
- Innerhalb der Vorlage können Kinds-Komponenten mit Inhalts-Knoten versehen werden
- `<user-list>`
 `<user-item>name</user-item>`
 `</user-list>`

Inhalte transkludieren

- Inhalts-Knoten werden transkludiert, wenn Vorlagen die **ng-content** Direktiven (Knoten) nutzen.
- Der Knoten stellt dabei einen Platzhalter da
- Mittels select Attribut lässt sich definieren, für welchen Inhalt der Platzhalter greifen soll

1

View

```
<user-list>
```

```
  <user-header></...>
```

```
  <user-item></...>
```

```
  <user-item></...>
```

```
  <user-item></...>
```

```
</user-list>
```

2

UserList - Template

```
<h3>user-list</h3>
```

```
<ng-content></...>
```

```
  <select="user-header">
```

```
    <user-item></...>
```

```
  </ng-content>
```

```
  <user-item></...>
```

```
</ng-content></...>
```

Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Decorator
 - `@ContentChild`
 - Parameter: Komponentenkasse
 - Optionales Options-Objekt mit Read-Eigenschaft
 - `{ read:`
 - `ElementRef | ViewContainerRef | Directive | Service }`

Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
- Zugriff erst nach Hook
 - `ngAfterContentInit`

Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
 - `@ContentChildren`
 - Parameter: Komponentenkasse
 - Erzeugt
 - `QueryList<Type>`
 - `changes` -> `Observable`

- *Erstelle eine user-header Komponente*
- *Erstelle eine user-item Komponente, die Inhalte nach der Zeichenkette ‚user:‘ transkludiert*
- *Nutze und Teste die Komponente in der Vorlage der user Komponente*

Übung 07 – Teil 1

- *Erweitere die user-list Komponente so, dass Sie item und header transkludiert*
- *Nutze die user-list Komponente entsprechend in der user Komponenten-Vorlage*
- *Ermittle in der user-liste die Instanzen der transkludierten Inhalte und gebe Sie aus (console)*

Übung 07 – Teil 2

Vorlagen Elemente ermitteln

- Über Eigenschafts-Decorator
 - `@ViewChild`
 - Parameter: Komponentenkasse / Hash-ID
 - Optionales Options-Objekt mit Read-Eigenschaft
 - `{ read:`
 - `ElementRef | ViewContainerRef | Directive | Service }`

Vorlagen Elemente ermitteln

- Über Eigenschafts-Dekorator
- Zugriff erst nach Hook
 - `ngAfterViewInit`

Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
 - `@ViewChildren`
 - Parameter: Komponentenkasse
 - Erzeugt
 - `QueryList<Type>`
 - `changes` -> `Observable`

- *Refactore das Beispiel: Die user Komponente soll nur noch die user-list als Kinds-Komponente habe*
- *Die user-list Komponenten-Vorlage soll user-header und user-item als Kinds-Komponente aufnehmen.*
- *Ermittle in der user-liste die Instanzen für user-header und user-item und gebe Sie aus (console)*

Übung 08

- *Erzeuge in der user-list ein hr-Element mit #Auszeichnung*
- *Ermittle das Element via ViewChild*
- *Ändere die Border-Color*
- *Zunächst über die style – Eigenschaft*
- *Anschließend über Renderer2*

Übung 09

Dynamische Inhalte erstellen

- Vorlagen dynamisch nutzen
 - Strukturelle Direktive: `*ngTemplateOutlet`
 - Vorlagen-Referenz
- `ViewContainerRef` (via DI)
 - `createEmbeddedView` - Methode
 - `templateRef : TemplateRef<T>`

- *Erzeuge in der user-list Vorlage via ng-template eine Template Referenz*
- *Nutze die Referenz in der Vorlage über die strukturelle Direktive ngTemplateOutlet*
- *Nutze als Direktiven Wert den Hash des Templates*

Übung 10 – Teil 1

- *Nutze die Template Referenz aus Übung 10 – Teil 1*
- *Ermittle in der user-list die Template-Referenz via ViewChild*
- *Injiziere in die user-Liste die ViewContainerRef*
- *Nutze diese Referenz um über createEmbeddedView eine Instanz der Template-Referenz anzuhängen.*

Übung 10 – Teil 2

Dynamische Inhalte erstellen

- Eigene Strukturelle-Direktiven lassen sich als Vorlagen-Referenz nutzen
- via `@ViewChild` oder `@ContentChild` kann die Direktive ermittelt werden
- mittels `read` Eigenschaft kann die Template Referenz der Direktive ermittelt werden
 - `@ContentChild (UserHeaderDirective, { read: TemplateRef })`

- *Nutze eine eigene Strukturelle Direktive zur Definition einer Template Referenz*
- *Ermittle in der user-list die Template-Referenz via ViewChild oder ContentChild und der read Option*
- *Nutze die Referenz, um über `ViewContainerRef.createEmbeddedView` eine Instanz anzuhängen*

Übung 11

Dynamische Komponenten erstellen

- Komponente in `entryComponents` aufnehmen
- Strukturelle Direktive: `*ngComponentOutlet`
 - Ausdruck zur Bindung einer Komponenten-Klasse
- Optionale Eigenschaften
 - `ngOutletInjector`, `ngOutletProviders` & `ngOutletContent`

- *Erzeuge eine neue Komponente user-dynamic und registriere sie in den entryComponents*
- *Nutze in user-list die Strukturelle Direktive ngComponentOutlet und bind einen Ausdruck zu der neuen Komponenten-Klasse*

Dynamische Komponenten erstellen

- Komponenten Factory erzeugen
 - `ComponentFactoryResolver` (DI)
 - `resolveComponentFactory` Methode
 - Parameter: Komponente
(muss in `entryComponents` sein)

Dynamische Komponenten erstellen

- Dyn. Komponente über Factory anhängen
- `ViewContainerRef` (via DI)
 - `createComponent` - Methode
 - `compFactory` : `ComponentFactory<T>`

- *Erzeuge eine Factory für UserDynamicComponent*
- *Nutze die ViewContainerRef und hänge eine UserDynamicComponent in den DOM*

Eltern-Kind-Kommunikation

Eltern-Komponente

```
export class UserListComponent {  
  userList: User[];  
  selectUser (user: User) {}  
}
```

```
<user  
  [userData]="userList[0]"  
  (onSelect)="selectUser($event)"  
>
```

Kind-Komponente

```
export class UserComponent {  
  @Input() userData: User;  
  @Output() onSelect: EventEmitter;  
}
```

Komponentenattribute

- Benutzerdefinierte Attribute lassen sich über den Eigenschaftsdekorator anlegen
 - `@Input (OPT_ATTR_NAME)` name: Type
- Auch für Setter nutzbar
- `ngOnChanges` : Hook informiert über neue Werte
 - `SimpleChanges`

- *Refactore das user-item und nutze statt der Transkludierung ein ,name' Attribut*
- *Refactor die ,items' Eigenschaft zu einem string[]*
- *Binde die Werte der items Liste über ngFor*
- *Hänge dich an onChangees und variiere die gebundenen Werte.*

Komponentenereignisse

- Benutzerdefinierte Ereignisse lassen sich über den Eigenschaftsdekorator anlegen
 - `@Output (OPT_ATTR_NAME)` name: `EventEmitter<T>`
- EventEmitter sendet Wert via emit
- Elter-Komponenten können sich an das Ereignis hängen
 - `$event` – Übertragener Ereigniswert

- *Die user-item Komponente soll ein Ereignis feuern, wenn darauf geklickt wird.*
- *Der Klick soll via @HostListener registriert werden.*
- *Das Ereignis soll via @Output für die user-list zugänglich gemacht werden*

- *Erzeuge eine eigene strukturelle Direktive, die via @Input einen roleTyp: string entgegennimmt*
- *Beschaffe dir via DI die TemplateRef und ViewContainerRef*
- *Das Template soll angezeigt werden, wenn es noch nicht existiert und die empfangene Rolle === provided ROLE.*
- *Für Schnelle: Definiere die Rolle in einer eigenen Environment-Datei*

Übung 16

Komponenten-Lebenszyklus

constructor

ngOnChanges

ngOnInit

ngDoCheck

ngAfterContentInit

ngAfterContentChecked

ngAfterViewInit

ngAfterViewChecked

ngOnDestroy

```
export class UserListComponent
```

```
<userList [data]="userList">
```

```
<userList>Vorlage
```

```
<user></user>
```

```
<user> </user>
```

```
</userList>
```

Änderungserkennung

- Wird bei jedem asynchronen Ereignis ausgeführt
- Dabei erfolgt die Erkennung in Kette
 - Eltern-View -> Kinds-View
- Hooks werden vor (init, change, doCheck) und nach (after) der Erkennung ausgeführt
- DOM Aktualisierung ist die Folge der Änderungserkennung

Änderungserkennung: Unter der Haube

- Die `ChangeDetectionRef` ist die Super-Klasse der `ViewRef`
- Jede Direktive und Komponente ist Bestandteil einer View
- Die View steuert somit die Änderungserkennung und DOM Aktualisierung
- Die `ChangeDetectionRef` lässt sich via DI injizieren

Änderungserkennung - Beeinflussen

- `ChangeDetectionRef`
 - `detach()`
 - Entfernt Komponente von der Änderungserkennung
 - Analog - `ChangeDetectionStrategy.OnPush`
 - `reattach()`
 - Hängt Komponente wieder in die Erkennung

Änderungserkennung - Beeinflussen

- ChangeDetectionRef
 - markForCheck()
 - Erkennung Eltern -> Kindern
- detectChanges()
 - selbst -> Kinder

Änderungserkennung - beeinflussen

- Änderungserkennung umgehen
 - *ngZone* (DI)
 - *runOutsideAngular*

Änderungserkennung - OnPush

- Änderungserkennung mit der Push Strategie
 - Wird durch asynchrone Ereignisse ausgelöst
 - Wird bei Wertwechsel einer Bindung ausgelöst
 - *async Pipe* – Unterstützt bei der Auslösung