ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Angular Components
Data-binding & Lifecycle Hooks

Alex Thalhammer

Outline

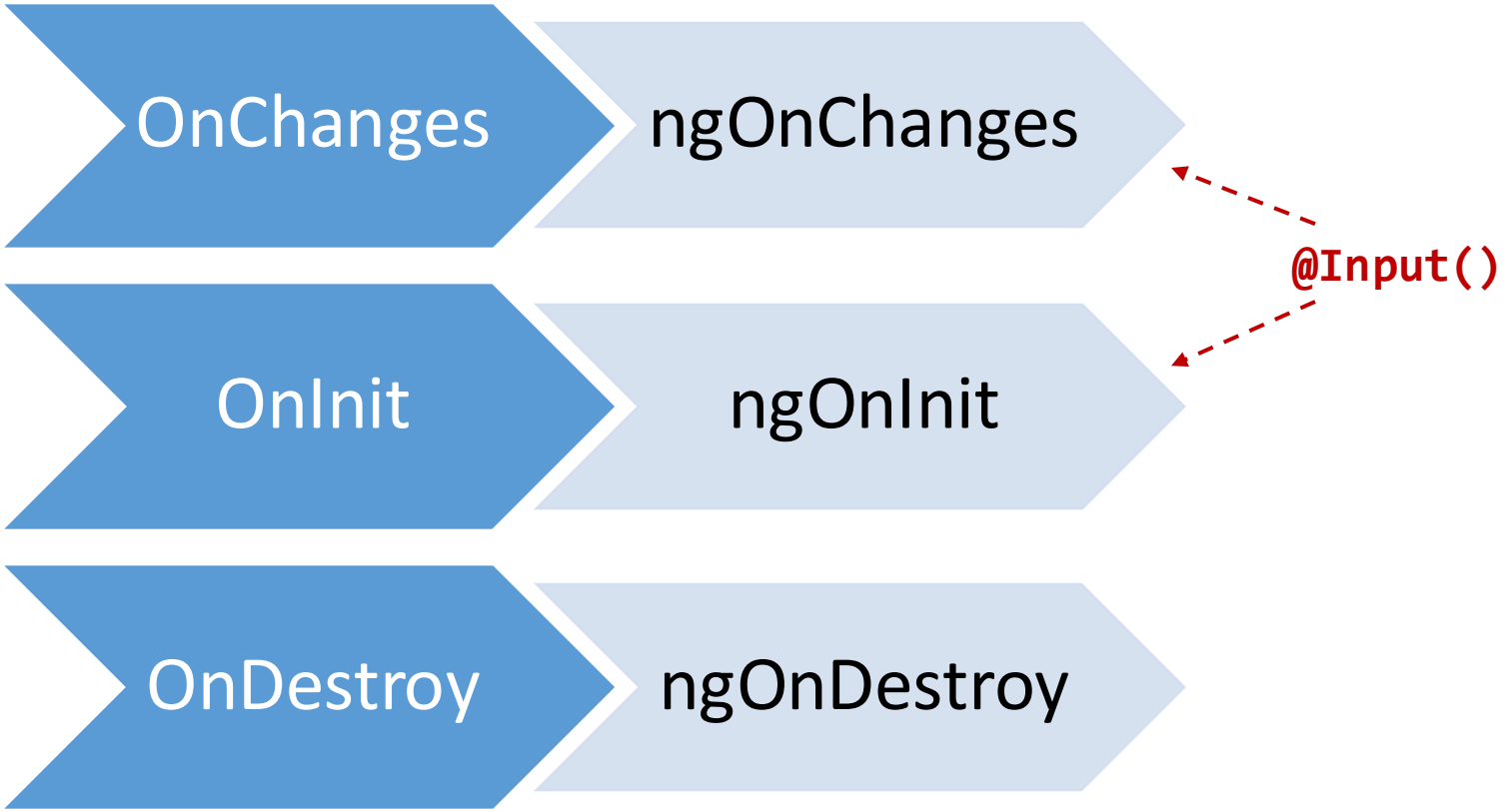
- Component Lifecycle Hooks
- Take a closer look on data binding
  - Property binding with @Input()
  - Event binding with @Output()
  - Two-way bindings
- View vs Content
  - ng content projection
- Smart vs dumb components

Lifecycle Hooks

What are Lifecycle Hooks?

- Built in methods in our components & directives
- Will be called at a certain time by Angular

Lifecycle Hooks (selection)



Usage

```
@Component({
    selector: 'my-component',
    […]
})
export class SomeComponent implements OnChanges, OnInit {

    @Input() someData;

    ngOnChanges(changes: SimpleChanges): void {
        […]
    }

    ngOnInit(): void {
        […]
    }
}
```

DEMO

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

Data binding

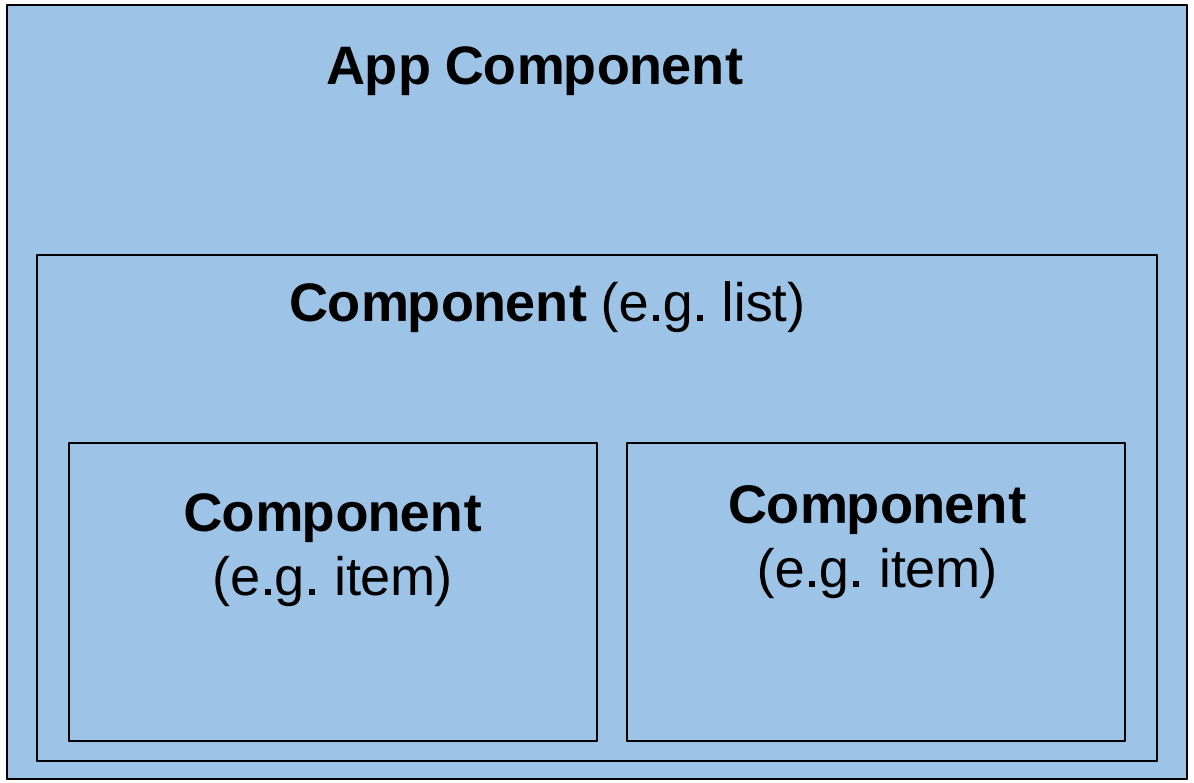
Performance

Components

Predictability

Architecture goals in Angular

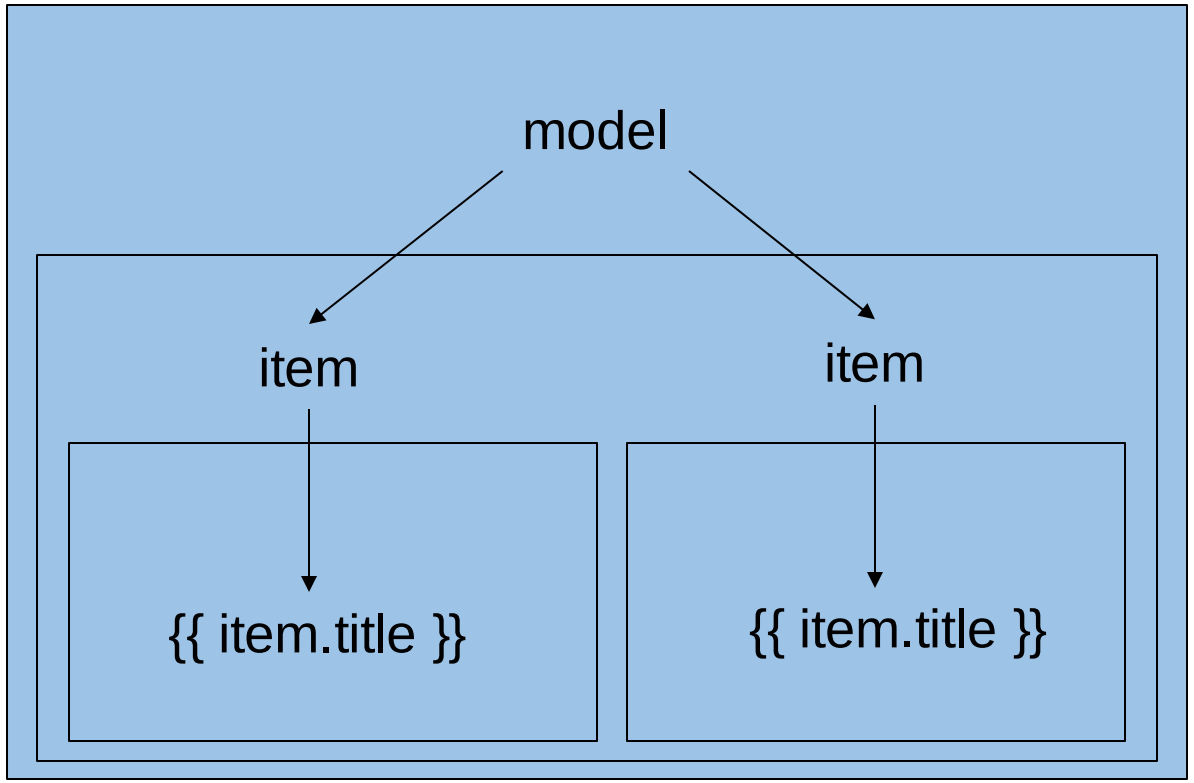
Component tree in Angular 2+



Rules for input/property binding []

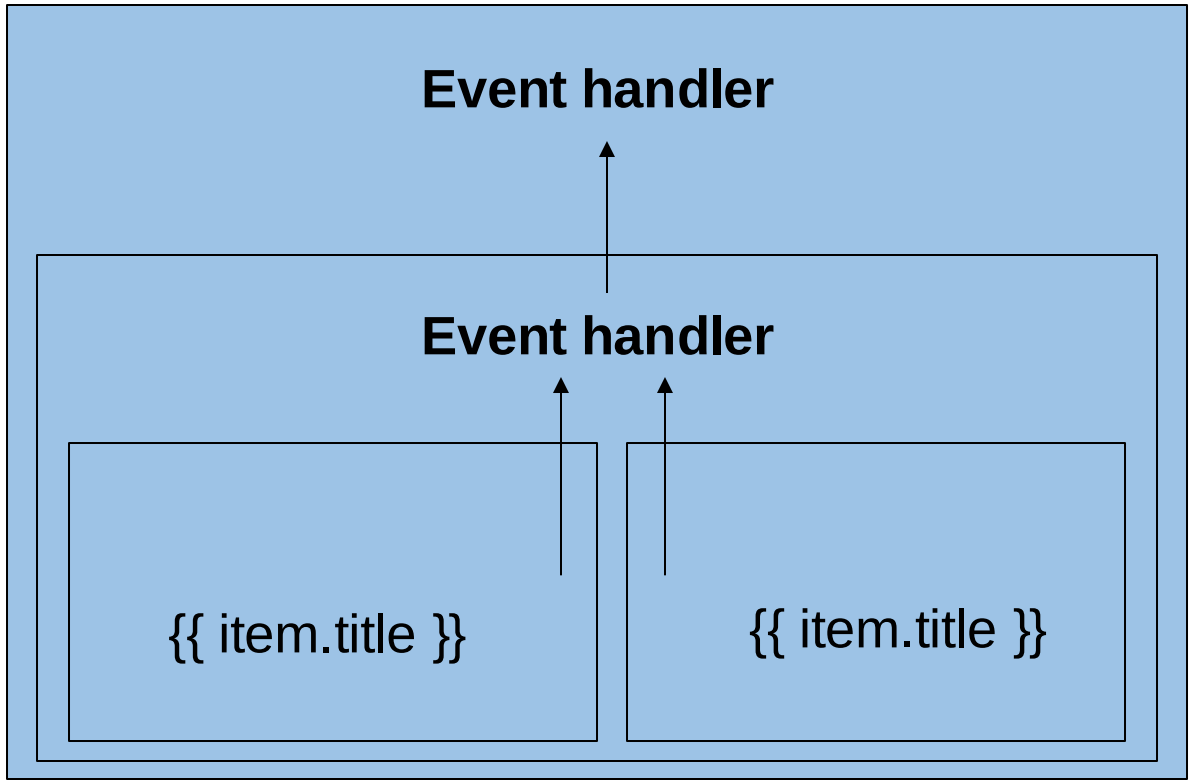
- Data can only be passed from top to bottom (top/down)
  - Parent can pass data to children
  - Children cannot pass data to parent (we need events for that)

Input/property binding []



[http://victorsavkin.com/post/110170125256/change-detection-in-angular-2]

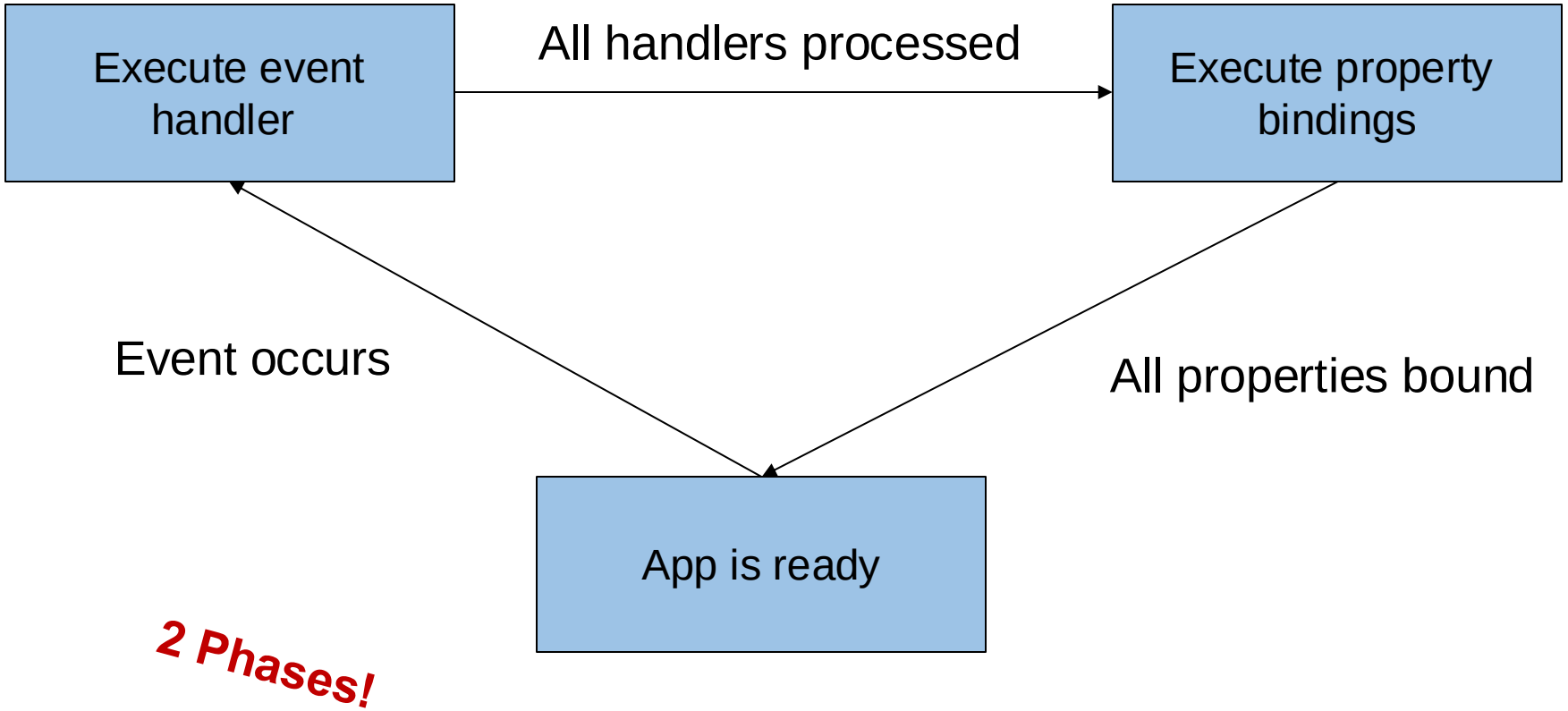
Ouptout/event bindings (one way bottom/up)



Event bindings (one way, bottom/up)

- Events can trigger data change → Property Binding

Property and event bindings



View

```
<button [disabled]="!from || !to" (click)="search()">
   Search
</button>

<table>
   <tr *ngFor="let flight of flights">
      <td>{{ flight.id }}</td>
      <td>{{ flight.date }}</td>
      <td>{{ flight.from }}</td>
      <td>{{ flight.to }}</td>
      <td><a href="#" (click)="selectFlight(flight)">Select</a></td>
   </tr>
</table>
```

```
<td [text-content]="flight.date"></td>
```

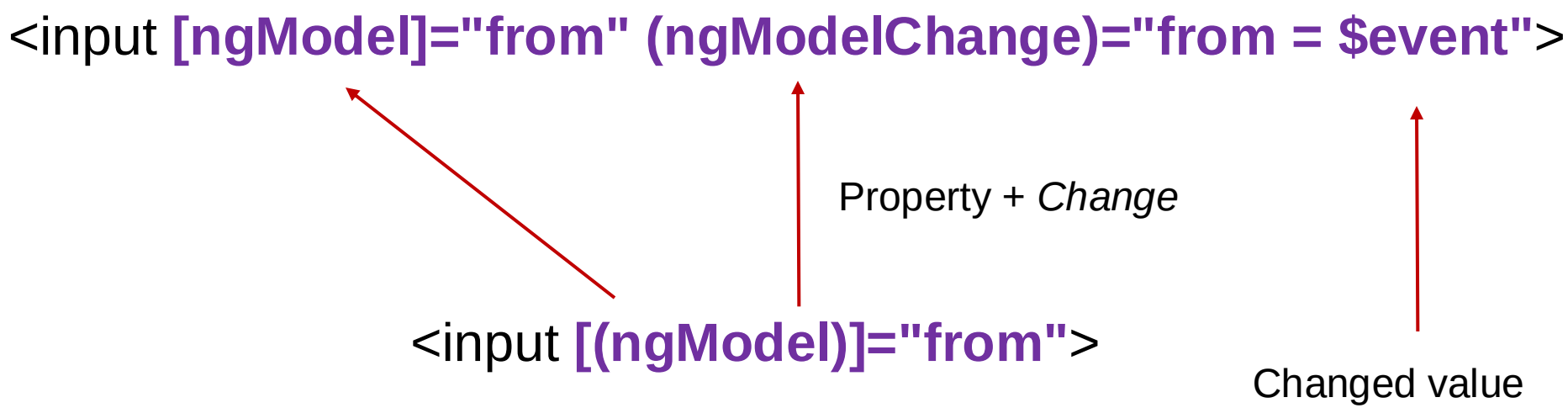
Recap

- Property binding: one way; top/down
- Event binding: one way; bottom/up
- Two way bindings?
- Two way = property binding + event binding

Property + event binding

```
<input [ngModel]="from" (ngModelChange)="update($event)">
```

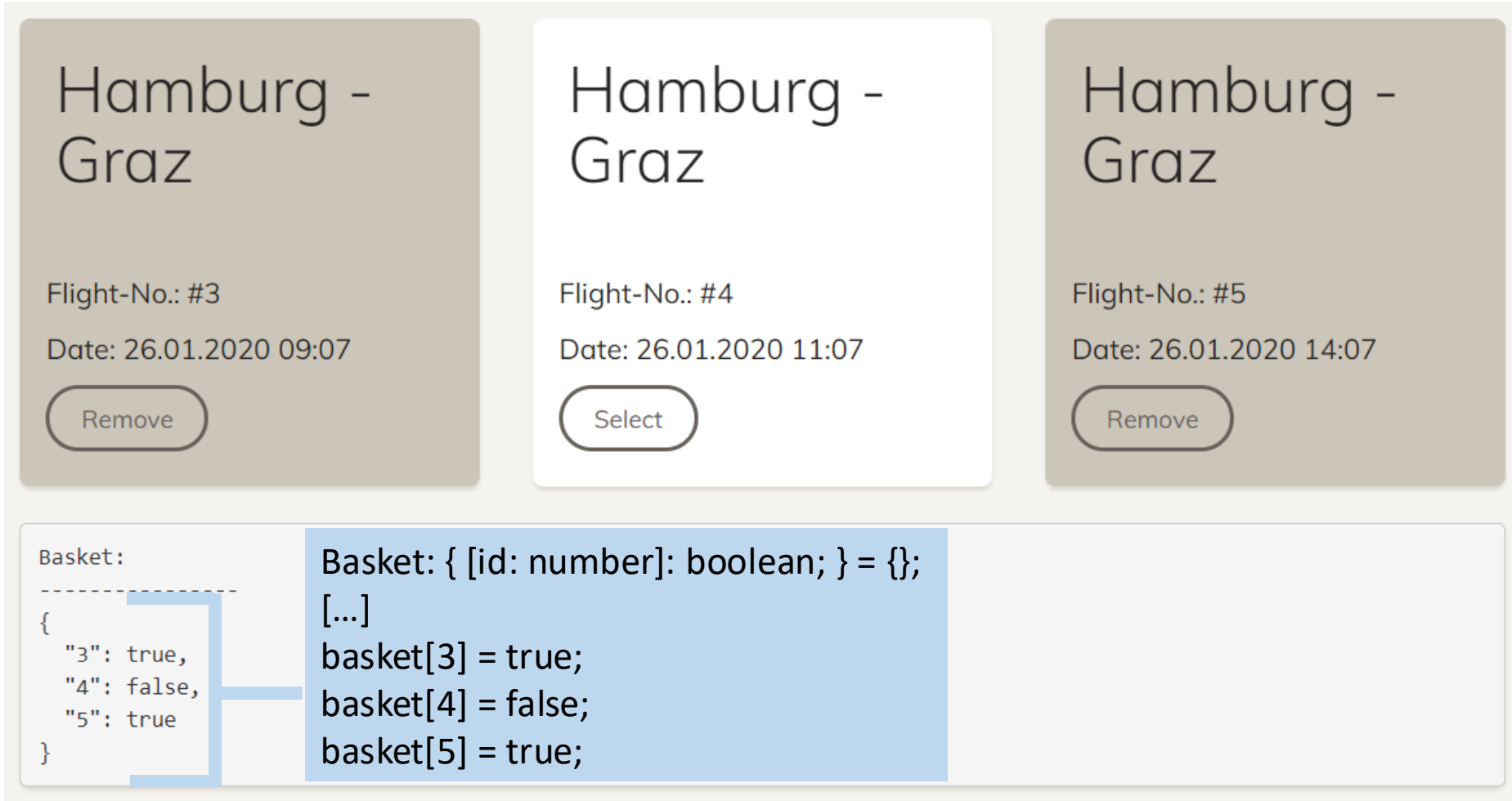
Property + event binding



Components data bindung

Example: app-flight-card

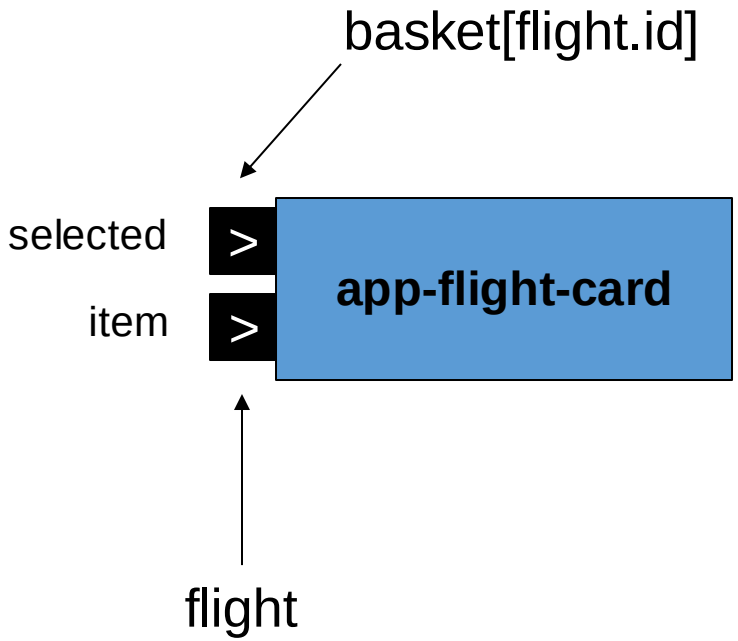
Example: app-flight-card



Example: app-flight-card in flight-search.html

```
@for (flight of flights; track flight.id) {
    <app-flight-card [item]="flight" [selected]="basket[flight.id]" />
}
```

app-flight-card



Example: app-flight-card

```
@Component({
    selector: 'app-flight-card',
    templateUrl: './flight-card.component.html'
})
export class FlightCardComponent {

    […]

}
```

Example: app-flight-card

```
export class FlightCardComponent {
    @Input({ required: true }) item!: Flight;
    @Input() selected = false;

    select(): void {
        this.selected = true;
    }

    deselect(): void {
        this.selected = false;
    }
}
```

View Template

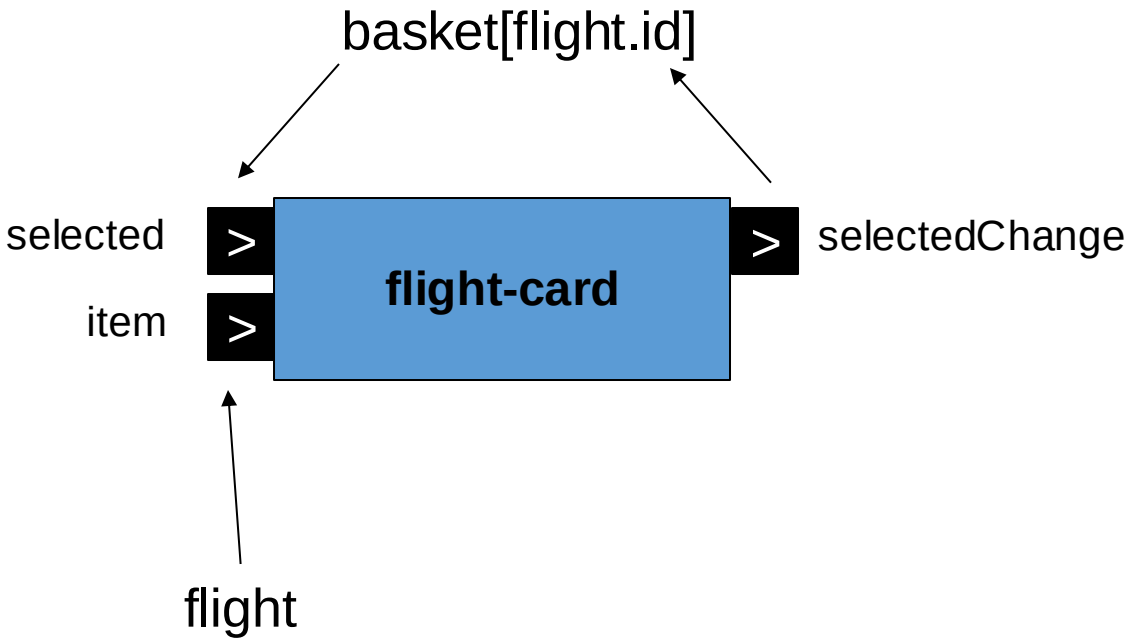
```
<div style="padding:20px" [class.is-selected]="selected">
    <h2>{{item.from}} - {{item.to}}</h2>
    <p>Flightnr. #{{item.id}}</p>
    <p>Date: {{item.date | date:'dd.MM.yyyy'}}</p>
    <p>
        @if (!selected) {
            <button (click)="onSelect()">Select</button>
        } else {
            <button (click)="onDeselect()">Deselect</button>
        }
    </p>
</div>
```

View Template – Alternative

```
<div style="padding:20px" [class.is-selected]="selected">
    <h2>{{item.from}} - {{item.to}}</h2>
    <p>Flightnr. #{{item.id}}</p>
    <p>Date: {{item.date | date:'dd.MM.yyyy'}}</p>
    <p>
        <button (click)="onToggleSelect()">
            {{ selected ? 'Deselect' : 'Select' }}
        </button>
    </p>
</div>
```

Event bindings

flight-card



Example: flight-card event *selectedChange*

```
@for (flight of flights; track flight.id) {
   <app-flight-card [item]="flight"
                    [selected]="basket[flight.id]"
                    (selectedChange)="basket[flight.id] = $event" />
}
```

Example: flight-ca

```
export class FlightCardComponent
        @Input({ required: true
        @Input() selected = fals
        @Output() selectedChange

        select(): void {
                this.selected = true;
                this.selectedChange.emit(this.selected);
        }

        deselect(): void {
                this.selected = false;
                this.selectedChange.emit(this.selected);
        }
}
```

```
<div *ngFor="let f of flights">
   <app-flight-card [item]="f"
                    [selected]="basket[f.id]"
           (selectedChange)="basket[f.id] = $event">
   </app-flight-card>
</div>
```

Example: flight-card event *two-way binding*

```
@for (flight of flights; track flight.id) {
  <app-flight-card [item]="flight" [selected]="basket[flight.id]" />
}
```

```
@for (flight of flights; track flight.id) {
  <app-flight-card [item]="flight" [(selected)]="basket[flight.id]" />
}
```

DEMO

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

LAB

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

View vs. Content

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

View vs. Content

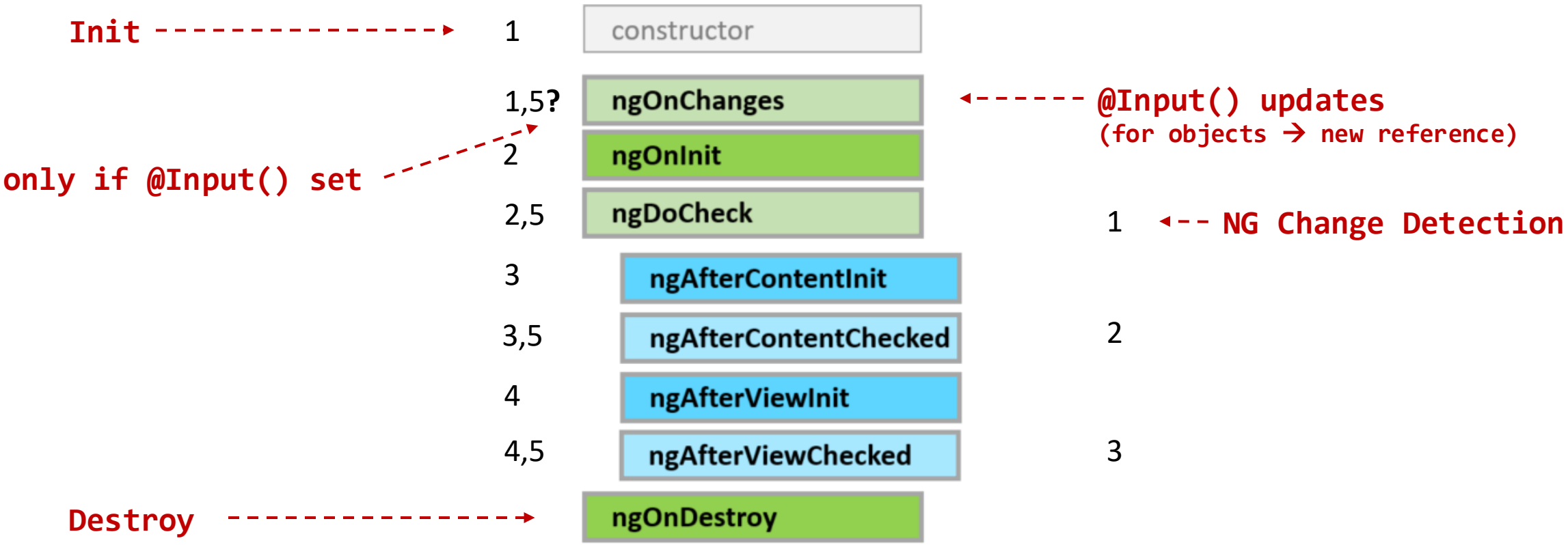
```
@Component({
selector: 'app-tab',
template: `
    @if (visible) {
        <h1>{{title}}</h1>
        <div>
            <ng-content>No content</ng-content>
        </div>
    }
`
})
export class TabComponent {
    @Input() title = '';
    visible = true;
}
```

View

```
<app-tab title="Booked">
    Sample Text ...
</app-tab>
```

Content

Lifecycle Hooks (in order of execution)



Thought experiment

- What if <app-flight-card> would handle use case logic?
  - e.g. communicate with API
- Number of requests ==> Performance?
- Traceability?
- Reusability?

Smart vs. Dumb Components

Smart / Controller	Dumb / Presentational
• 1 use case / route • Business logic • Container	• Independent of Use Case • Reusable • Often Leafs

LAB

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

Signal Components

(Starting with Angular 17.1)

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Signal Based Components (V17.1)

```
@Component({
  selector: 'app-temperature',
  template: `
    <p>C: {{ celsius() }}</p>
    <p>F: {{ fahrenheit() }}</p>
  `,
})
export class TemperatureComponent {
  celsius = signal(0);
  fahrenheit = computed(() => this.celsius() * 1.8 + 32);
}
```

Signal Inputs (readonly, V17.1)

```
@Component({
  selector: 'app-temperature',
  template: `
    <p>C: {{ celsius() }}</p>
    <p>F: {{ fahrenheit() }}</p>
  `,
})
export class TemperatureComponent {
  // celsius = input(0); // InputSignal<number>
  // celsius = input.required(0); // InputSignal<number>
  celsius = input.required<number>(); // InputSignal<number>
  fahrenheit = computed(() => this.celsius() * 1.8 + 32);
}
```

```
<!-- parent component template -->
<app-temperature [celsius]="25" />
```

Why Signal Inputs

- Will automatically mark **OnPush** components as dirty
- Type safety
  - Required inputs do not require initial values! ☺
    - or tricks to tell TypeScript that an input always has a value
  - Transforms are automatically checked
- Values can be easily derived upon changes using **computed()**
- Monitoring thru **effect()** instead of **ngOnChanges()**

Signal Outputs (V17.3)

```
@Component({…})
export class TemperatureComponent {
  tempChange = output<number>(); // OutputEmitterRef<number>

  // or
  tempChangeSubject = new Subject<number>();
  tempChange = outputFromObservable(this.tempChangeSubject); // OutputEmitterRef<number>
}
```

```
<!-- parent component template -->
<app-temperature (tempChange)="handleTempChange($event)" />
```

Why Signal Outputs

- The API is conceptually **aligned** with the new APIs for signal inputs
- Simpler API and removed complexity that isn’t relevant for outputs
- Automatic clean-up of outputs upon destruction (no unsubscribing)
- Improved type safety for new outputs & classical EventEmitters

Signal Model (V17.3)

```
@Component({…})
export class TemperatureComponent {
  temperature = model(0); // writeable InputSignal<number> & OutputEmitterRef<number>
}
```

```
<!-- parent component template -->
<app-temperature [(temperature)]="myTemperature" />
```

```
<!-- parent component template with signal -->
<app-temperature [temperature]="temperature" (temperatureChange)="updateTemperature($event)" />
```

Signal Based Components

Work w/o Zone.js

Interop with traditional components

Fine-grained CD

No need to update code!