Trivadis AG
Sägereistrasse 29
CH-8152 Glattbrugg
training@trivadis.com

Angular Component Workshop

Vers. 1.0.0 / Oktober 2018

Autoren:
Thomas Huber, Thomas Gassmann, Thomas Bandixen

Course Content

00_Course Introduction

Why a Workshop about Angular Components

- Components are a core concept of Angular
- There are plenty of ways how to communicate with Components
- With a some tricks, the code, testability and performance can be improved

trivadis
makes IT easier.

Your Trainer

Thomas Gassmann

Senior Consultant, Trainer, Speaker

thomasgassmann.net
@gassmannT
thomas.gassmann@trivadis.com

ANGULAR @trivadis

SWISS ANGULAR

trivadis makes IT easier.

Your Trainers

- Thomas Claudius Huber (@thomasclaudiush)
  www.thomasclaudiushuber.com
  thomas.huber@trivadis.com

- Thomas Bandixen (@tbandixen)
  thomas.bandixen@trivadis.com

- Thomas Gassmann (@gassmannT)
  www.thomasgassmann.net
  thomas.gassmann@trivadis.com

trivadis makes IT easier.

Course Hours

- Official course hours
  - 09:00 – 10:30
  - 10:45 – 12:00
  - 13:00 – 14:30
  - 15:00 – 17:00
- If you need a break in between, feel free to ask

trivadis makes IT easier.

Course Planning

- **Angular Component Workshop**
  - Introduction to Angular Components
  - Communicating with a Template
  - Input and Output Decorators
  - ViewChild and ViewChildren Decorators
  - Communication through a service
  - Architecture
  - Angular Elements

trivadis makes IT easier.

Introduce Yourself

- Name and job description
- Experience with Angular / TypeScript / JavaScript
- Experience with VSCode
- Expectations for this course

trivadis makes IT easier.

Your machine

- Visual Studio Code
  - Microsoft's powerful cross-platform code editor
- Node.js and NPM
  - powerful server and package manager
- Sample-code:
  - Structured like modules in this course
  - Available on
  - https://github.com/AngularAtTrivadis/AngularComponentWorkshop

trivadis makes IT easier.

Your machine

- Install Angular CLI

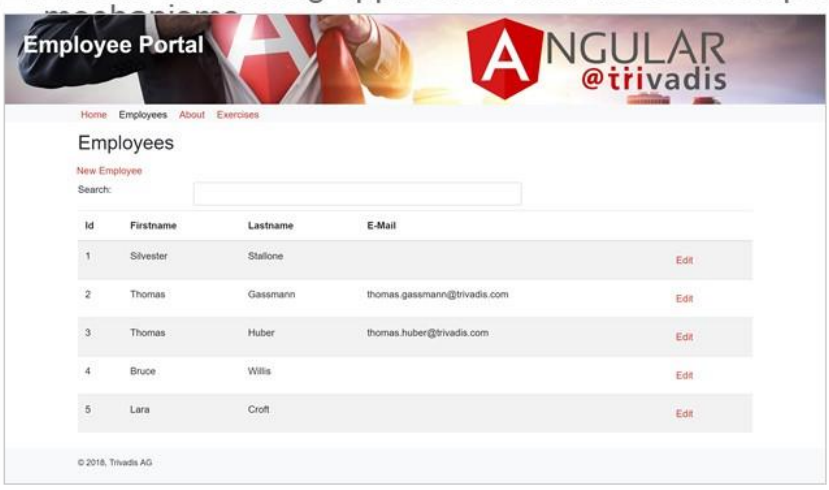
```
npm install -g @angular/cli
```

- VS Code Extension
  - Angular@Trivadis VS Code Essentials
  - https://marketplace.visualstudio.com/items?itemName=trivadis.ngtvd-extensions

trivadis makes IT easier.

What are we building?

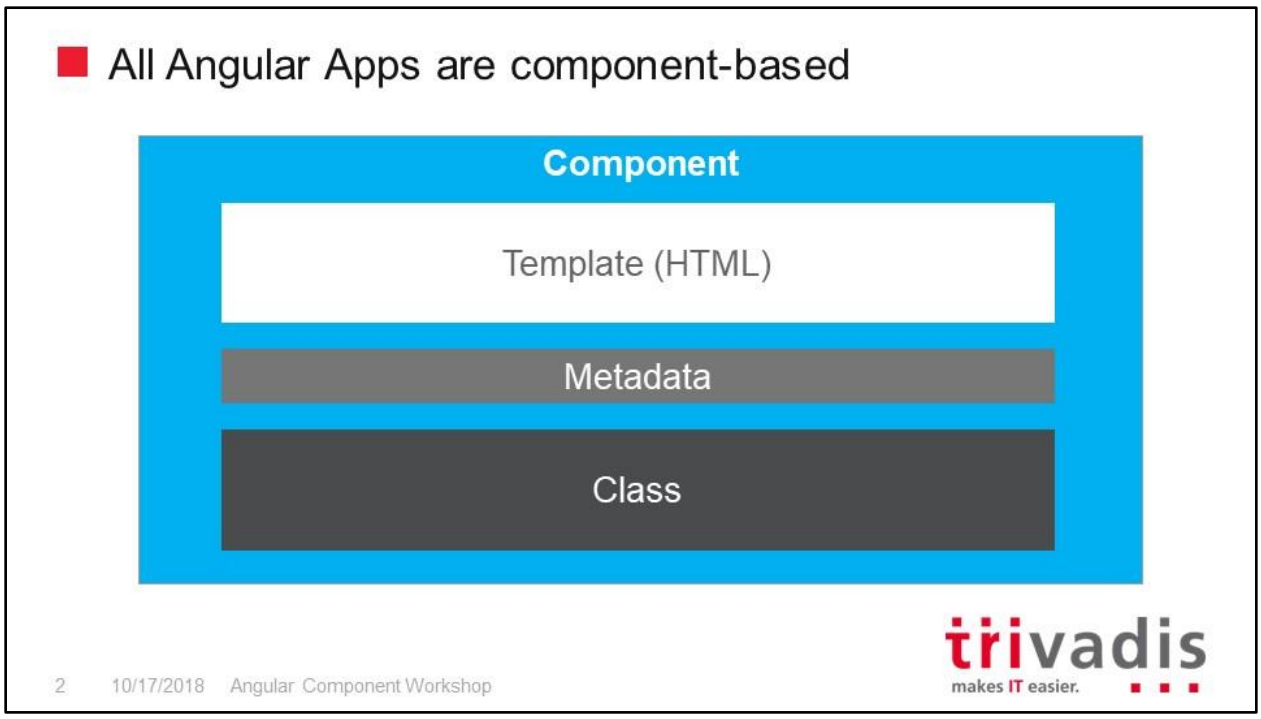
- Extend an existing application with different component mechanisms



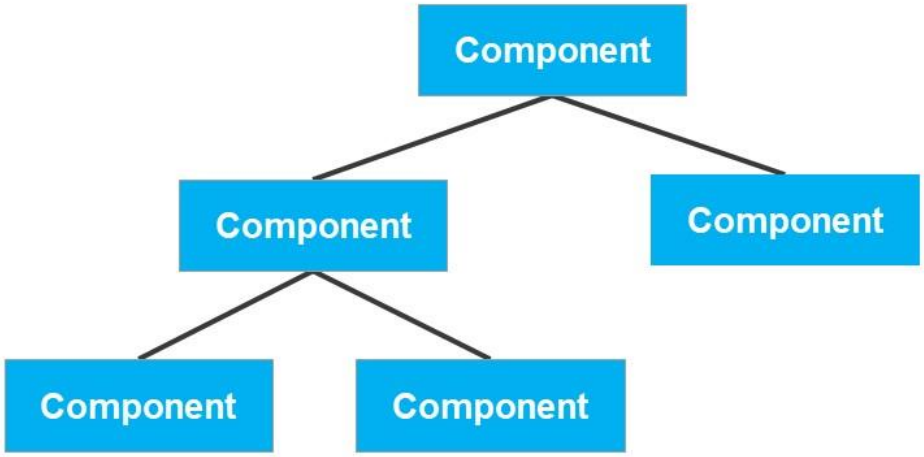
trivadis makes IT easier.


Let’s get our hands dirty!
11 10/17/2018 Angular Component Workshop
trivadis
makes IT easier.

01_Angular Components



An Angular app is a tree of components



trivadis makes IT easier.

Define a PersonDetailComponent

AppComponent

- Displays a list of persons
- Displays the details of the selected person
- Lets move the detail-display to a separate component

PersonDetailComponent
(person-detail.component.ts)

```
CLI: ng g component person-detail
```

trivadis makes IT easier.

Define a PersonDetailComponent

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

@Component({
        selector: 'person-detail',
        template: ` ...
             <input id="firstname" type="text"
               [(ngModel)]="person.firstname"/> ...`
    }
)
export class PersonDetailComponent {
    person: Person;
}
```


Declare it in the AppModule

```
...
import {PersonDetailComponent} from './person-detail.component';

@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [AppComponent, PersonDetailComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Use it in the AppComponent

- Just use the chosen selector «person-detail»

```
<person-detail></person-detail>
```


trivadis makes IT easier.

Ways to define a Component's template

- Inline via template-property

```
@Component({
  selector: 'my-app',
  template: '<h1>Hello</h1>'
})
```

- Linked via templateUrl-property

```
@Component({
  selector: 'my-app',
  templateUrl: '/app/app.component.html'
})
```


trivadis makes IT easier.

Ways to define a Component's styles

- Inline via styles-property

```
@Component({
  selector: 'my-app',
  styles: ['.isSelected{ color:green }']
})
```

- Linked via styleUrls-property

```
@Component({
  selector: 'my-app',
  styleUrls: ['/app/app.component.css']
})
```


Data Binding overview

- **{{}}** => Display the value of a property in the UI
- **[]** => Bind an element-property to a property of your class
- **()** => Bind an element-event to a method of your class
- **[()]** => Bind a property two-way
  - that you don't have to think whether it's **([])** or [()], just use the "banana in a box"-mnemonic to get the syntax right

Simple Rendering with Interpolation

- Use an expression in {{ }}
  - for example {{1 + 1}} will write out 2
- use the pipe iterator to pass the expression to a Pipe
  - you can write your own pipes or use existing pipes like *uppercase*

```
@Component({
  selector: 'my-app',
  template: `<h1>Details of {{fullname | uppercase}}</h1>`
})
export class AppComponent {
  fullname: string = "Lara Croft";
}
```


Binding properties

- use the brackets [] to bind to a property
- This creates a OneWay-Data Binding

```
@Component({
  selector: 'my-app',
  template: `<h1>Details of {{fullname | uppercase}}</h1>
             <input type="text" [value]="fullname">`
})
export class AppComponent {
  fullname: string = "Lara Croft";
}
```

Binding events

- Use parentheses () to bind events to methods

```
@Component({...,
  template: `...
            <input type="text" [value]="fullname">
            <button (click)="onUpdate()">Update</button>`
})
export class AppComponent {
  fullname: string = "Lara Croft";
  onUpdate()
  {
    this.fullname = "Duke Nukem";
  }
}
```


Add the FormsModule to your AppModule

- This allows you to use ngModel for TwoWay-Data Bindings

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {FormsModule} from '@angular/forms';
import { AppComponent }   from './app.component';

@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

TwoWay-Data Bindings

```
<input type="text"
  [ngModel]="fullname"
  (ngModelChange)="fullname = $event">`
```

- You can see above that neither a DOM-property nor a DOM-event is involved => it's pure Angular
- As Angular knows the bound property and the bound event, it has a special shortcut syntax: «the banana in the box»

```
<input type="text"
  [(ngModel)]="fullname">
```


TwoWay-Data Bindings: Binding to an object

- Just use the dot-syntax to access properties

```
@Component({
  selector: 'my-app',
  template: `<h1>Details of {{person.fullname | uppercase}}</h1>
            <input type="text" [(ngModel)]="person.fullname">`
})
export class AppComponent {
  person: Person = { fullname: "Lara Croft" };
}

interface Person { fullname:string }
```

Structural Directives

- **ngFor** – to generate elements while looping over a collection
- **ngIf** – to show or hide an element with an if-statement
- **ngSwitch** – to show/hide elements based on a condition

trivadis makes IT easier.

ngFor

```
@Component({...
  template: `...
            <table>
              <tr *ngFor="let person of persons">
                <td>{{person.firstname}}</td>
                <td>{{person.lastname}}</td>
                <td>{{person.githubaccount}}</td>
              </tr>
            </table>
             `
})
export class AppComponent {
   persons:Person[]=PERSONS;
}
```


trivadis makes IT easier.

ngIf

- The div is only displayed if selectedPerson is not undefined/null

```
@Component({...
  template: `...
            <div *ngIf="selectedPerson">
              ...
            </div>
          `
})
export class AppComponent {
   persons:Person[]=PERSONS;
   selectedPerson:Person; ...
}
```


ngIf

- Since Angular 4 ngIf else can be used for loading messages

```
@Component({...
  template: `...
            <div *ngIf="selectedPerson; else loading">
              ...
            </div>
            <ng-template #loading>Loading…</ng-template>
          `
})
export class AppComponent {
   persons:Person[]=PERSONS;
   selectedPerson:Person; ...
}
```

ngSwitch

- Based on the firstname a div is displayed

```
<div [ngSwitch]="selectedPerson?.firstname">
    <div *ngSwitchCase="'Thomas'">I'm teaching Angular</div>
    <div *ngSwitchCase="'Lara'">I'm playing games</div>
    <div *ngSwitchDefault>I'm just a fallback</div>
</div>
```


About stars (*) and templates

- ngFor, ngIf and ngSwitch use a * for the attribute – that's Angular's syntactical sugar
- all these directives add/remove a subtree within a template tag
- the template-tag gets generated behind the scences

About stars (*) and templates

- This is what Angular does with an *ngIf

```
<input *ngIf="selectedPerson" .../>
```

```
<input template="ngIf:selectedPerson" .../>
```

```
<ng-template [ngIf]="selectedPerson">
  <input .../>
</ng-template>
```


trivadis makes IT easier.

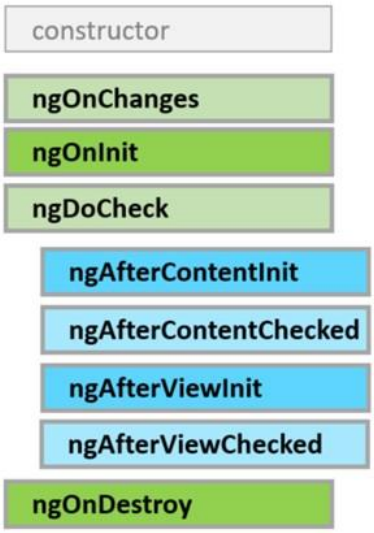
Component Lifecylce Hooks

- You can hook into the life cycle of your component. Just implement these interfaces:
  - **OnInit** - intialize your component and load data
  - **OnChanges** - react to changes of input properties
  - **OnDestroy** – perform cleanup actions

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

@Component( ... )
export class PersonDetailComponent implements OnInit {
    ngOnInit() { console.log('initialized'); }
}
```


trivadis makes IT easier.

Component Lifecylce Hooks



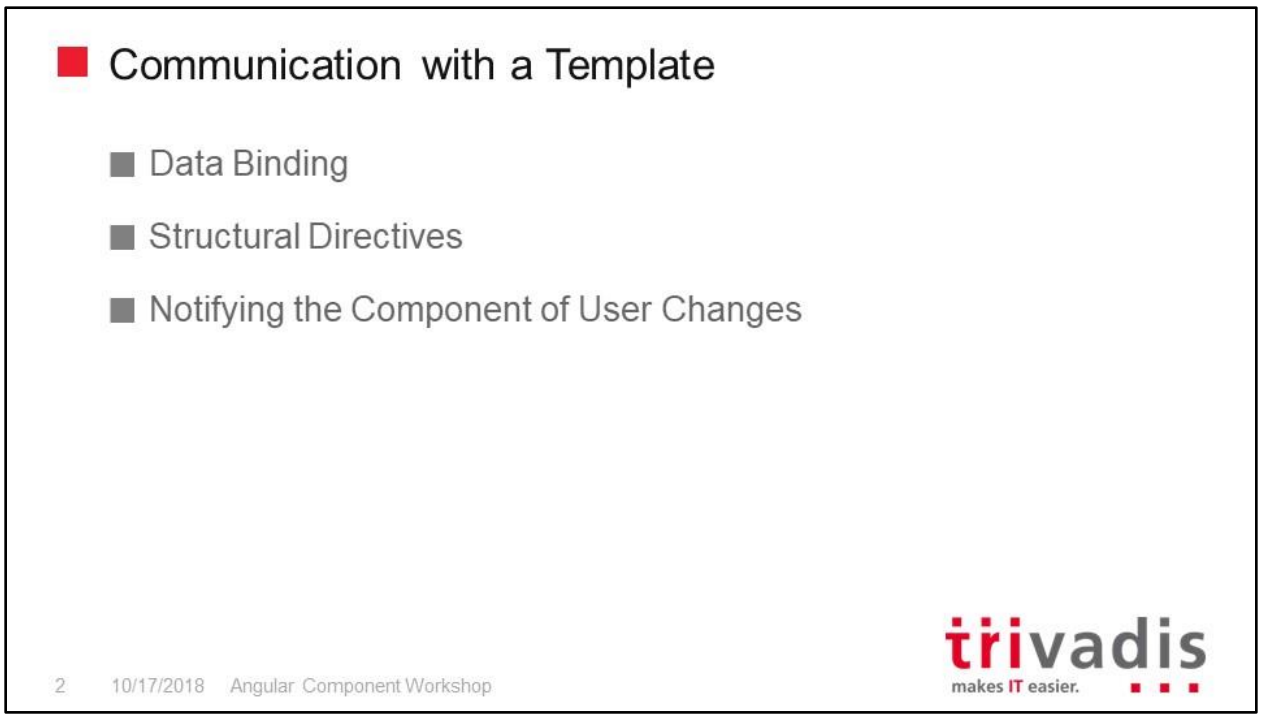
trivadis makes IT easier.

Summary

- Components need to be declared in an app module
- Components have their own HTML-selector
- Lifecycle Hooks are used by implementing their interface

trivadis makes IT easier.

02_Communicating with a Template



Notifiyng the Component of User Changes

- Two-way Databinding with ngModelChange
- Getter/Setter in Two-way Databinding
- valueChanges observable

trivadis makes IT easier.

Two-way Databinding with ngModelChange

- Normal two-way Databinding

```
<input type="text" [(ngModel)]="listFilter" />
```

- Two-way Databinding, the long form

```
<input type="text" [ngModel]="listFilter"
                    (ngModelChange)="listFilter=$event" />
```

- To interact or notify to the change, add a method

```
<input type="text" [ngModel]="listFilter"
                    (ngModelChange)="onFilterChange($event)" />
```


trivadis makes IT easier.

Getter/Setter in Two-way Databinding

- Getter and Setter

```
private _listFilter: string;

get listFilter: string {
  return this._listFilter;
}
set listFilter(value: string) {
   this._listFilter = value;
   // perform filtering or any other action
}
```


trivadis makes IT easier.

Angular Forms

Template-Driven Forms

- Angular creates the form
- Defined in the template
- Access references with ViewChild

Reactive Forms

- We create the form structure
- Define in the component
- No need for ViewChild

trivadis makes IT easier.

ValueChanges observable: Template-Driven

- Even if input is not inside a form, we can access it via ViewChild

```
<input type="text" [(ngModel)]="listFilter" />
```

```
@ViewChild(NgModel) filterInput: NgModel;
…
this.filterInput.valueChanges.subscribe(() =>
   this.performFilter(this.listFilter)
);
```


ValueChanges observable: Reactive Froms

- Subscribe directly to the form control

```
this.filterInput.valueChanges.subscribe(() =>
   this.performFilter(this.listFilter)
);
```

Summary

- Whenever possible, use bindings and structural directives
- They are simple, common and easy to understand for any other developer on the team
- Avoid using two-way Databinding the long way.
- Getter and Setters are a good alternative but require more lines of code
- Or subscribe to the valueChanges event

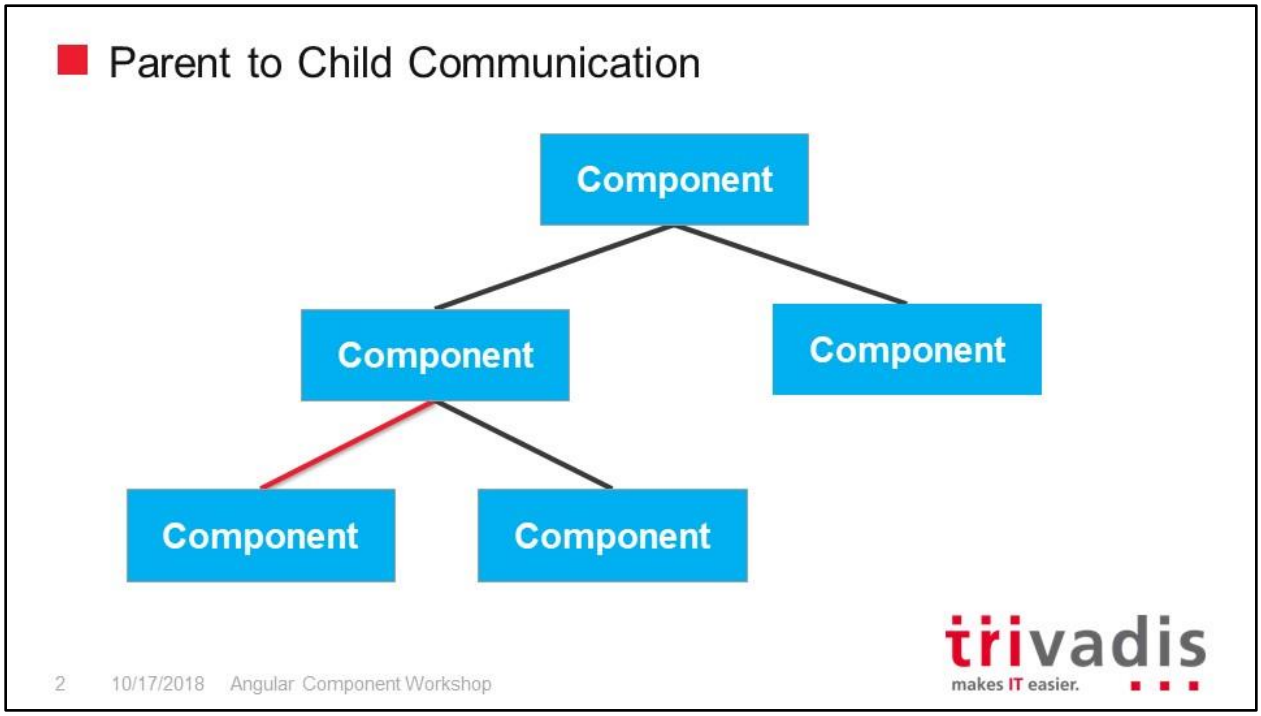
trivadis makes IT easier.

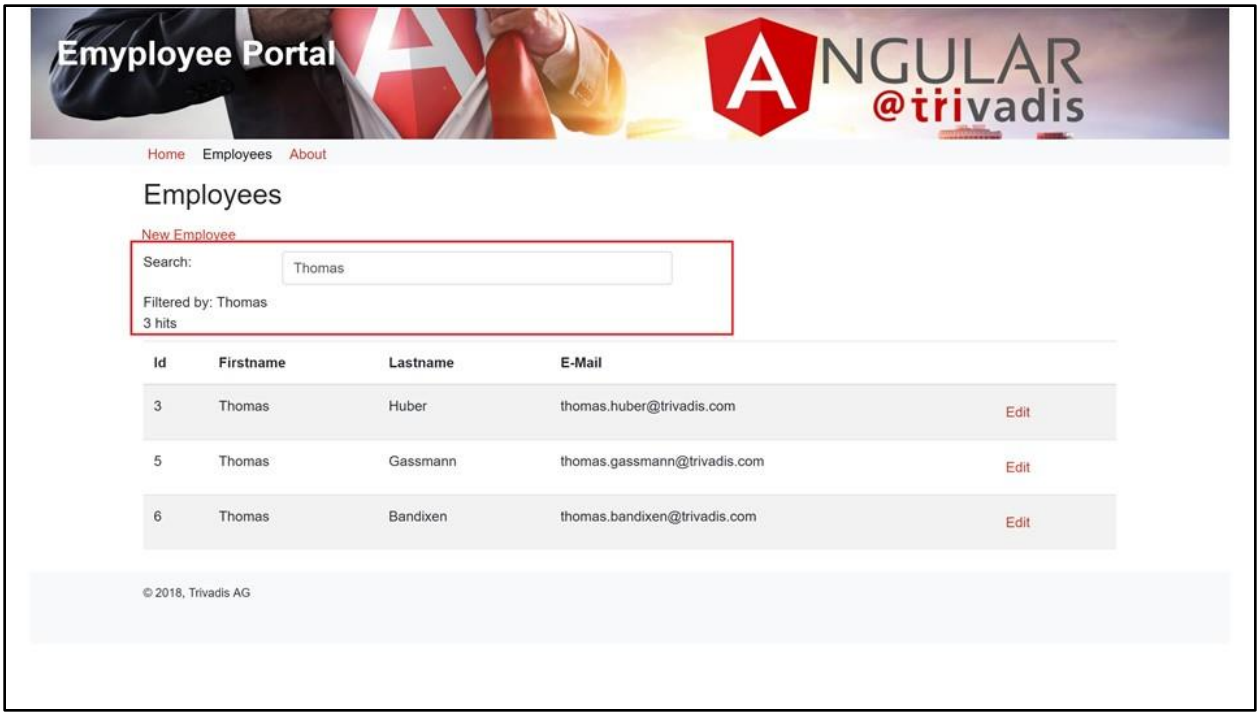
Change listFilter input to the long way of two-way Databinding

Change it to Getter/Setters

trivadis makes IT easier.

03_Input and Output Decorators





Parent Child Communication Use Cases

Using an Input property

- Configure the child component (e.g. enable status)
- Provide a default value (e.g. past the last filtered criteria)
- Provide an item data to the child (e.g. edit form)

Request information with a ViewChild decorator or Template Reference Variable

- Poll data from a child
- Tell the child to perform an action

trivadis
makes IT easier.

Use the Input of a Component

- Existing code «person-detail»

```
<person-detail></person-detail>
```

- Bind the person-property to the selected person

```
<person-detail [person]="selectedPerson"></person-detail>
```

- This leads to an error. The person-property is only available in the PersonDetailComponent's template => Change it

The Input of a Component

- Use the @Input decorator
  - to make a components property available to the outside
  - this means that you can bind to the property

```
import { Component, Input } from '@angular/core';
import { Person } from './person';

@Component( ... )
export class PersonDetailComponent {
    @Input()
    person: Person;
}
```

The Input of a Component

```
@Component( ... )
export class PersonDetailComponent {
    @Input()
    person: Person;
}
```

- With the input defined, this works:

```
<person-detail [person]="selectedPerson"></person-detail>
```


Watching for Changes to an Input Property

- Getter and Setter

```
private _hitCount: Number;
get hitCount(): Number {
   return this._hitCount;
}
@Input()
set hitCount(value: Number) {
   this._hitCount = value;
   // action to do
}
```

Watching for Changes to an Input Property

- OnChanges Lifecycle Hock

```
@Input()
hitCount: number;

ngOnChanges(changes: SimpleChanges) {
   if(changes['hitCount'] && !changes['hitCount'].currentValue) {
        this.message = 'No matcvhes found';
   } else {
        this.message = 'Hits: ' + this.hitCount;
   }
}
```


trivadis makes IT easier.

Referencing a Child Component

- The parent needs a property from a child component
- Or wants to call a child components method
- Add a Template Reference Variable

```
<app-filter #filterComp
  [hitCount]="employees?.length" >
</app-filter>
```

- Now access any information from the child component

```
{{ filterComp.listFilter }}
```


trivadis makes IT easier.

Child to Parent communication use cases

Provide events with an Output decorator

- Event notification (e.g. parent needs to know when the filterValue changed)

Child component can provide properties and methods and expect the parent component to access it

- Child has data which it wants to provide to the parent (e.g. message for display)

Defining Component events

- Use Angular's EventEmitter

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

@Component( ...)
export class FilterComponent {
    ...
    valueChange = new EventEmitter<string>();

    onValueChange() {
       this.valueChange.emit(this.filterValue);
    }
}
```

The Output of a Component

- To use the event from the outside, define it as Output!

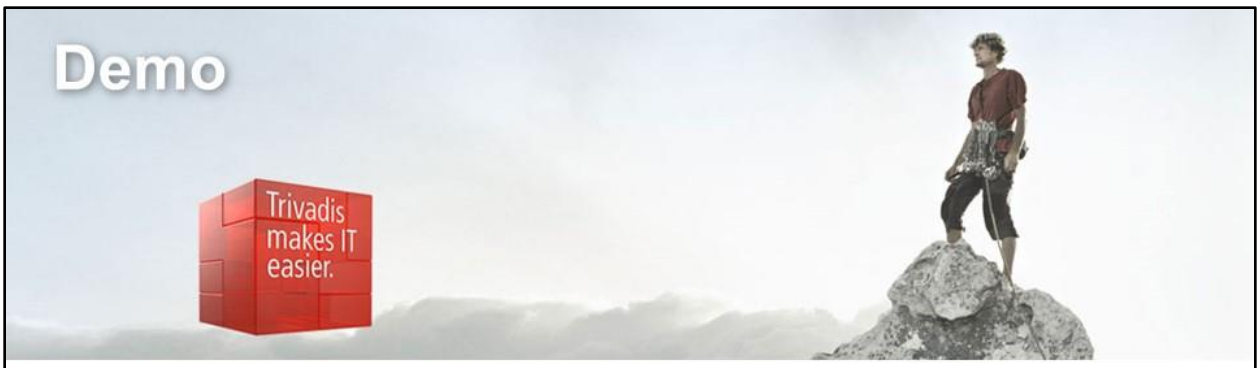
```
import { ..., Output, EventEmitter } from '@angular/core';

@Component( ... )
export class FilterComponent {
    ...
    @Output()
    valueChange = new EventEmitter<string>();
}
```

```
<app-filter [hitCount]="employees?.length"
  (valueChange)="onValueChanged($event)"></app-filter>
```


How to notify when the value change

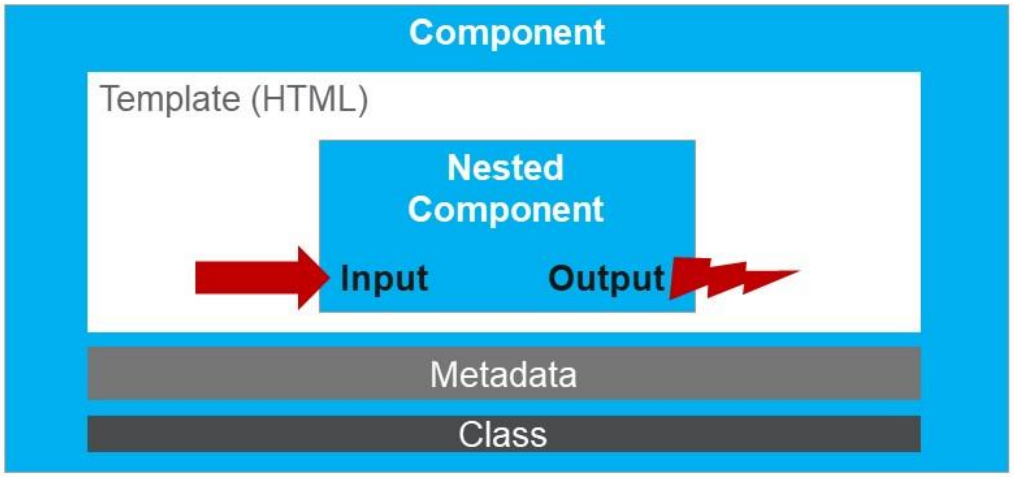
- For example with Getter/Setter

```
private _filterValue: string;
get filterValue(): string {
   return this._filterValue;
}
set filterValue(value: string) {
   this._filterValue = value;
   this.valueChange.emit(this._filterValue);
}
```



trivadis
makes IT easier.

Wrapup: Input and Output

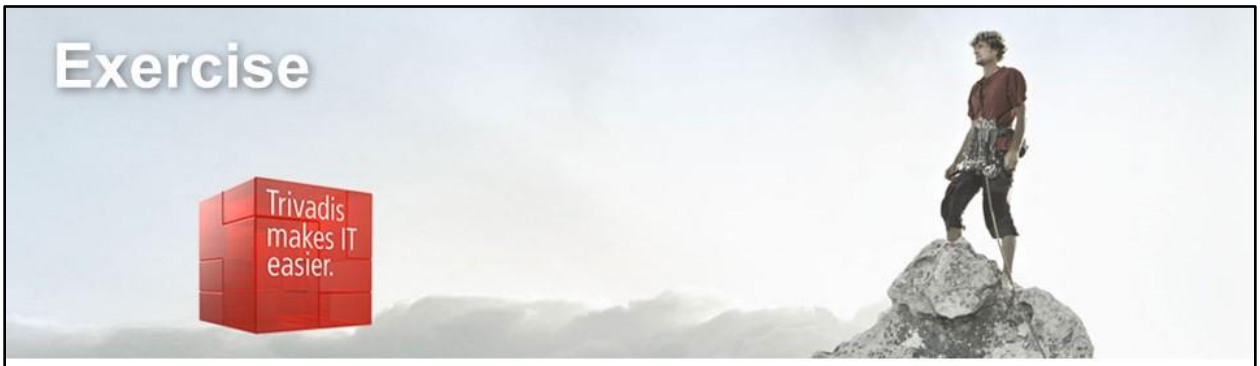


trivadis
makes IT easier.

Summary

- Define the «API» of your component with Input / Output
- Request information with a ViewChild decorator or Template Reference Variable
- Getter / Setter can be used to notify user input changes

trivadis makes IT easier.



Create a new filter component under shared/

Move the code to the new filter component

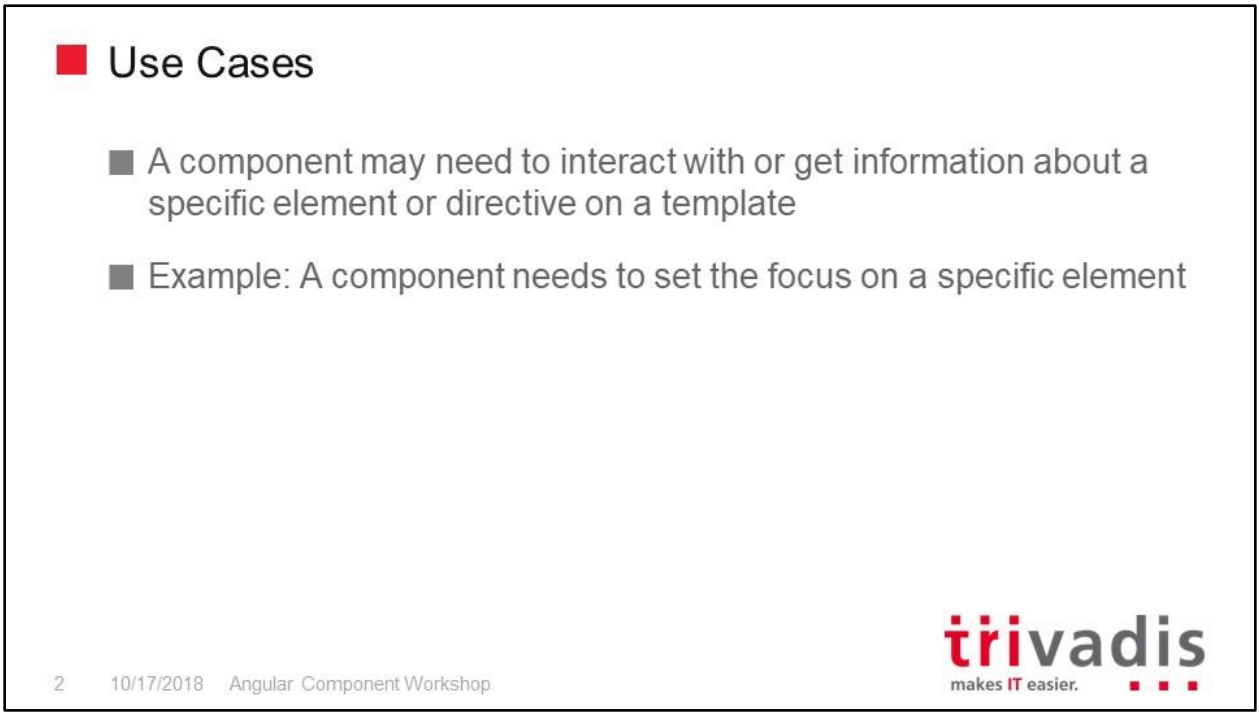
Add Input for hitCount

If hitCount is 0 show a different message (eg."no matches found")

Notify parent component when filter has changed

trivadis makes IT easier.

04_ViewChild_ViewChildren



Getting a Reference

- Javascript "old-school" way to access an element

```
let el = document.getElementById('myElId');
```

- In Angular we use the ViewChild decorator

```
@ViewChild('myEl') elementRef;
```


ViewChild

- The selector can be an Angular Directive

```
@ViewChild(ngModel) filterInput: ngModel;
```

- Custom directives

```
@ViewChild(MyComponent) myComp: MyComponent;
```

- Template Reference Variable

```
@ViewChild('myEl') elementRef;
```

ViewChild with Template Reference Variable

- Template Reference Variable

```
@ViewChild('filterEl') filterEl;
```

- Template

```
<input type="text" #filterEl [(ngModel)]="listFilter">
```

- Use it on the AfterViewInit Lifecycle Hook. View has to be initialized and rendered first

```
ngAfterViewInit(): void {
    this.filterEl.nativeElement.focus();
}
```


ViewChildren

- Accessing multiple elements or directives on a Template

```
@ViewChildren('myEl') divElementRefs: QuerySelector<ElementRef>;
```

- Tracks changes in the DOM

```
this.divElementRefs.changes.subscribe( () => {  …  });
```

- Same selectors as ViewChild but can have a set of variables

```
@ViewChildren('myEl', 'nameEl')
divElementRefs: QuerySelector<ElementRef>;
```

ViewChild and *ngIf

- When accessing an element inside *ngIf, it can be undefined
- Getter / Setter are a good solution
- Note: if subscribing inside the setter make sure to only subscribe once.
  Example: hold the subscription inside a variable

trivadis makes IT easier.

Summary

- ViewChild/Children provides a native element property
- Or provides a reference to the directive's data structures
- Only available after the AfterViewInit Lifecylce Hook
- It has to be inside the DOM to evaluate it

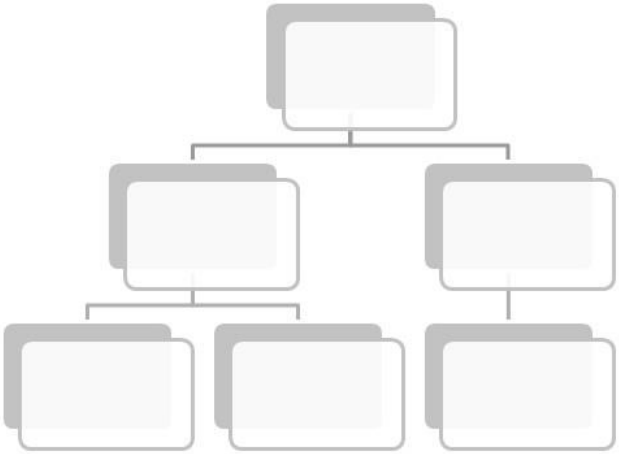
trivadis makes IT easier.

Add the focus on the search field with a Template Reference Variable and using the nativeElement

trivadis
makes IT easier.

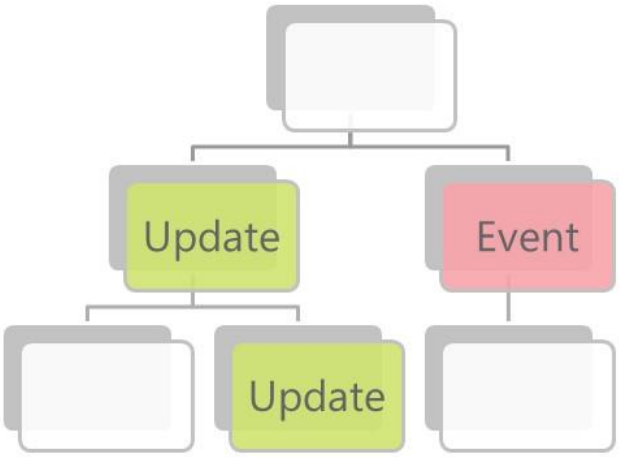
05_Communication through a Service

Typical Angular Application



trivadis makes IT easier.

Typical Angular Application



trivadis makes IT easier.

What is State?

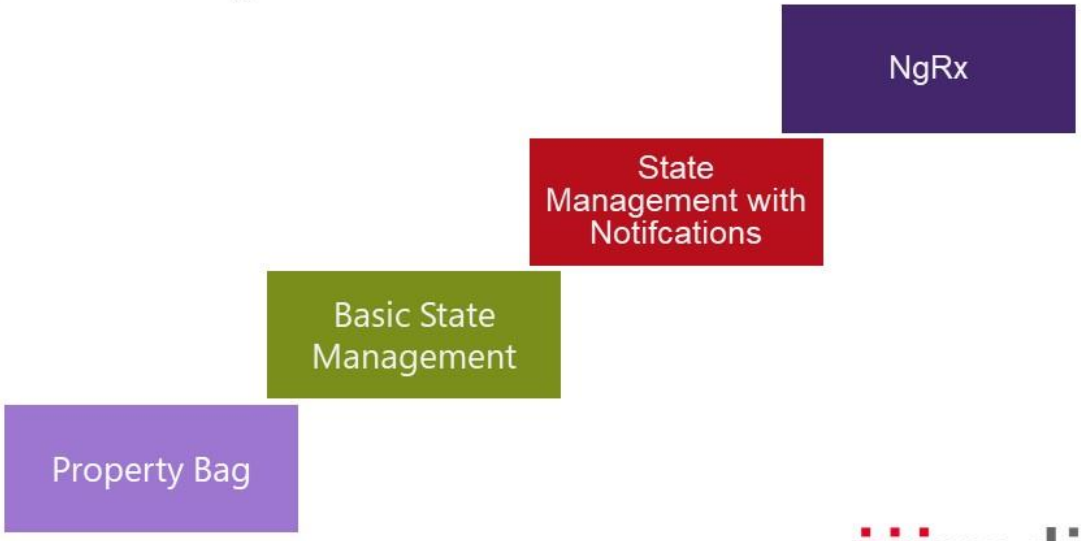
- View State
- User Information
- Entity Data
- User Selection and Input
- …

State Management

With a state management system we would like to have:

- A single-source of truth of our applications state and
- encapsulate our business logic from our view logic

State Management



trivadis makes IT easier.

Property Bag

- A service with a property of values
- Service holds a bag of property values
- The component has getters and setters which interact directly with the service
- We always loose the properties from a component when we navigate away

trivadis makes IT easier.

Retain the search values

- Scenario: A user enters a search, navigates to details and returns back => the search term is gone.
- Can also be done by QueryParams. Normally the better way.
- However, what if the search is huge and complex?
- Property Bag Service is a good alternative to QueryParams

Param Service

- A simple service

```
@Injectable()
export class ParamService {
  keyword: string;
}
```

- Component uses getter and setters

```
get listFilter() : string {
  return this.paramService.keyword;
}
set listFilter(value: string) : void {
  this.paramService.keyword = value;
}
```

Services

- Provide functionality across components
  Logging, Calculations, Data access, Data sharing, etc.
- Register it with the Angular Injector
- Inject into any component that needs it
- Normally, services are implemented as Singletons

trivadis makes IT easier.

Service Scope and Lifetime

- Services can be registered on the component or module level
- Service scope: Defines which components can see and access the service
- The service lifetime depends on where the service is registered

trivadis makes IT easier.

Register in a Component

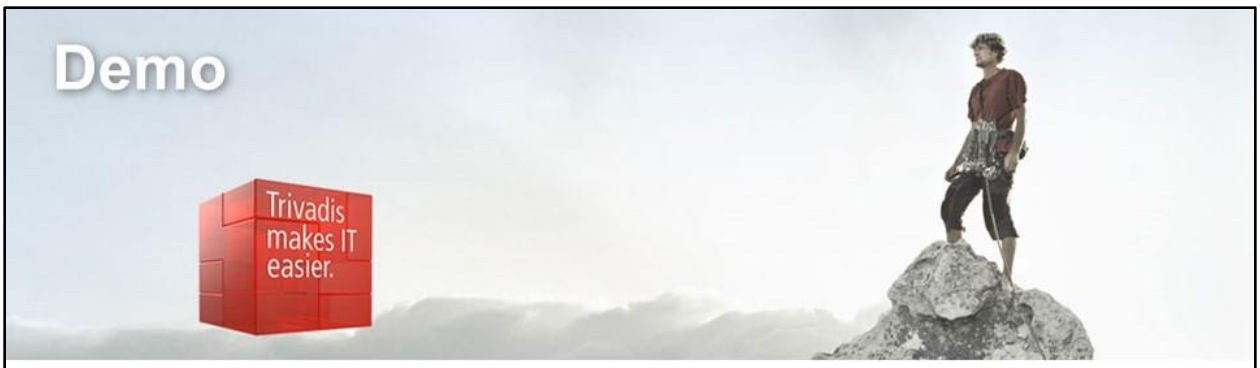
- The service is available inside the component and all children
- If we register the service in different components (not child component) we get different instances

```
@Component({
  templateUrl: 'employee-list.component.html',
  providers: [ EmployeeService ]
})
export class EmployeeListComponent implements OnInit { … }
```


Register in a Module

- The service is available in every component in the application
- If registered in a lazy module it behaves differently. It is only available inside the module.

```
@NgModule({
  imports: [SharedModule, EmployeeRoutingModule],
  exports: [],
  declarations: [...],
  providers: [...fromServices.services]
})
export class EmployeeModule {}
```



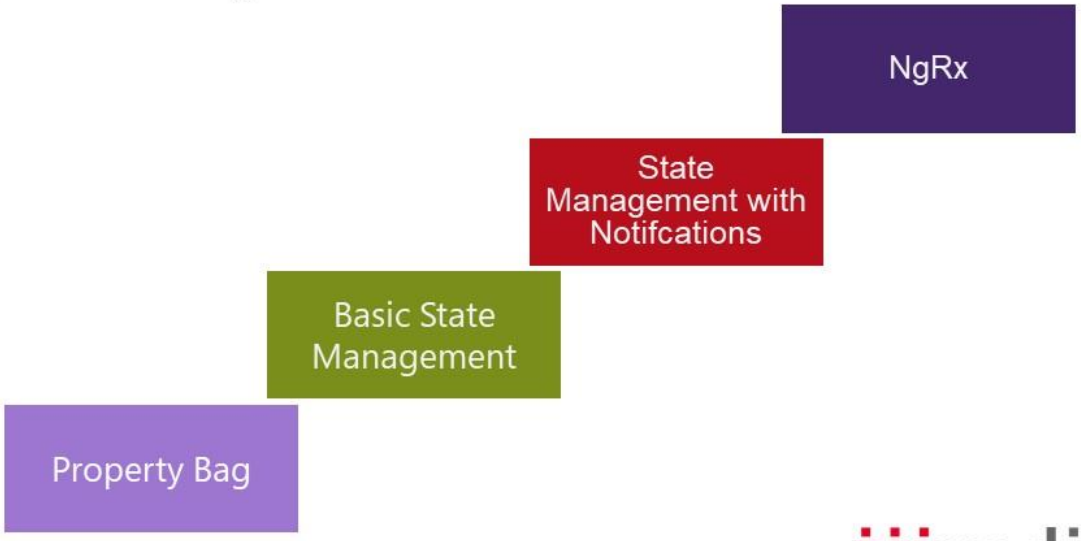
trivadis
makes IT easier.

Summary: Property Bag

- Is great for retaining view state and user selections
- Also for sharing data and communicating state changes
- But any component can read the values and change it
- Components are only notified of state changes if they use template binding

trivadis
makes IT easier.

State Management



trivadis makes IT easier.

Basic State Management Service

- Retrieving Entity Data (CRUD)
- Provide state values
- Data does not change very often in the backend, so do not load it every time
- Data is stored locally
- Observe state changes

trivadis makes IT easier.

Data Sharing Service

- Only load data the first time. Then use local instances

```
@Injectable()
export class EmployeeService {
  private employees: Employee[];

  constructor(private http: HttpClient) {}

  getEmployees() {}
  getEmployee(id: number) {}
  createEmployee(payload: Employee) {}
  …
}
```


Data Sharing Service: GetAll

- Only load data the first time. Then use local instances

```
getEmployees(): Observable<Employee[]> {
   if (this.employees) {
     return of(this.employees);
   }

   return this.http.get<Employee[]>(`URL`).pipe(
      tap(data => (this.employees = data)),
      catchError((error: any) => _throw(error))
   );
 }
```

Data Sharing Service: GetById

- Get instance from local array

```
getEmployee(id: number): Observable<Employee> {
    if (this.employees) {
      const item = this.employees.find(e => e.id === id);
      if (item) {
        return of(item);
      }
    }
    return this.http.get<Employee>(`URL`)
      .pipe(catchError((error: any) => _throw(error)));
  }
```


Data Sharing Service: Create

- Add it to the list if no error occurs

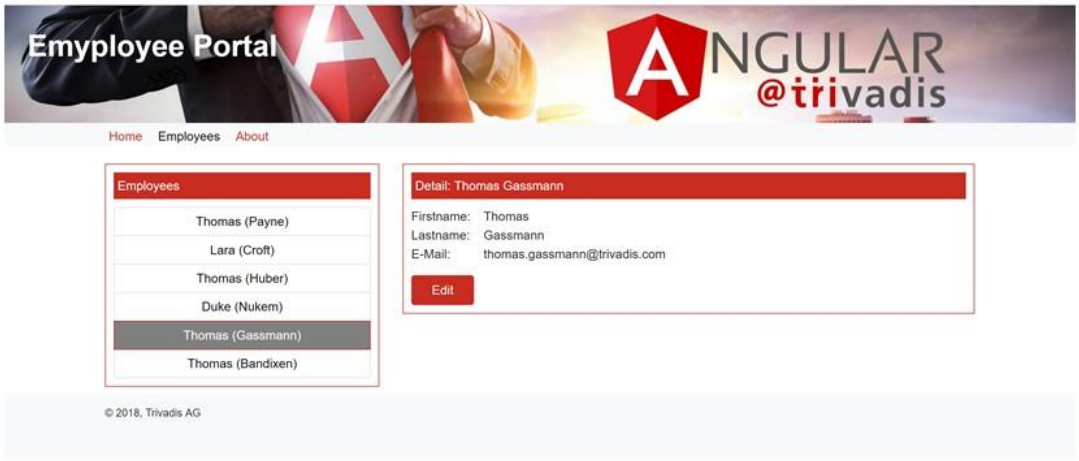
```
createEmployee(payload: Employee): Observable<Employee> {
   return this.http.post<Employee>(`URL`, payload)
      .pipe(
        tap(data => this.employees.push(data)),
        catchError((error: any) => _throw(error))
      );
  }
```

Data Sharing Service: Delete

- Delete locally if no error occurs

```
removeEmployee(payload: Employee): Observable<any> {
    return this.http.delete<any>(`URL`).pipe(
      tap(data => {
         const idx = this.employees.findIndex(e => e.id ===
                                                payload.id);
        if (idx >= 0) {
          this.employees.splice(idx, 1);
        }
      }),
      catchError((error: any) => _throw(error))
    );
  }
```


Keeping State in Sync



Track Selected Employee

- A property with the selected employee.

```
@Injectable()
export class EmployeeService {
  private employees: Employee[];
  public currentEmployee: Employee | null;
  …
}
```


Track Selected Employee

- A property with the selected product

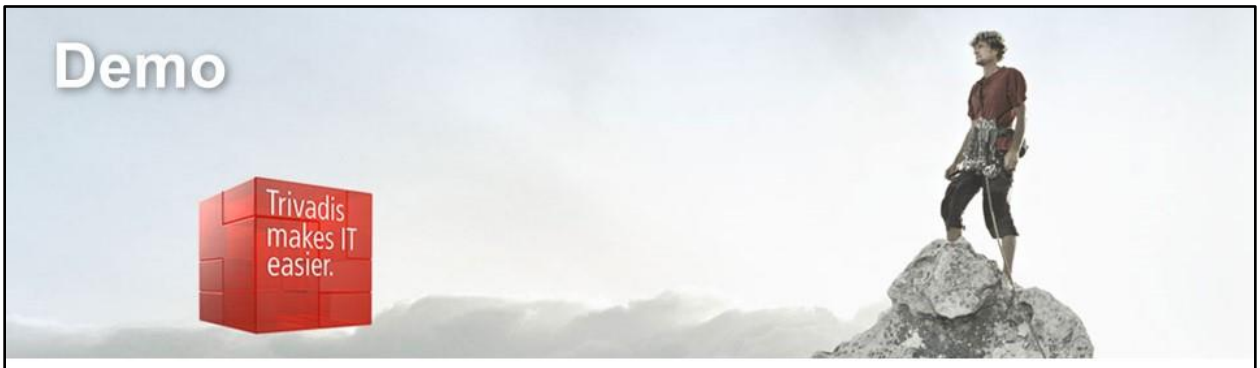
```
onSelected(employee: Employee) {
  this.employeeService.currentEmployee = employee;
}
```

Get Selected Employee

- Notify detail about the selected employee. A simple property does not work. For change detection a getter is appropriate. If everything is properly bound, it works automatically (pull).

```
@Component({
  templateUrl: 'employee-detail.component.html',
})
export class EmployeeDetailComponent implements OnInit {
   get employee(): Employee | null {
      return this.employeeService.currentEmployee;
   }
   …
}
```


trivadis makes IT easier.



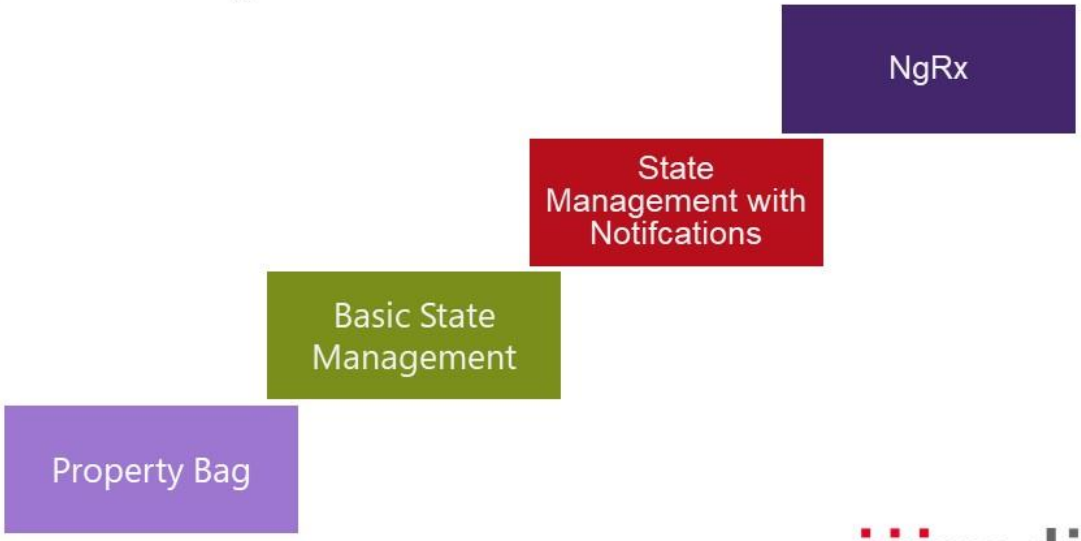
trivadis makes IT easier.

Summary: Basic State Management

- State Management Service for retrieving, managing and storing state
- Share between components
- Retains and shares state values
- Minimizes hits to the backend server
- Provides changes notifications for bound values using a getter
- No explicit change notification
- State is not immutable

trivadis makes IT easier.

State Management



trivadis makes IT easier.

State Management with Notifications

- Add notifications to the service
- Broadcasting service notifications
- EventEmitter is similar but not recommended
- Can be done with Subjects
- Or with BehaviorSubjects

Subject

- The key purpose is to send out notifications
- Is a special type of an observable which can multicast an event to multiple subscribers
- It is also an observer

```
this.mySubject.next(item);
```

Use the Subject

- Change the onSelected method

```
onSelected(employee: Employee) {
   this.employeeService.changeSelectedEmployee(employee);
}
```

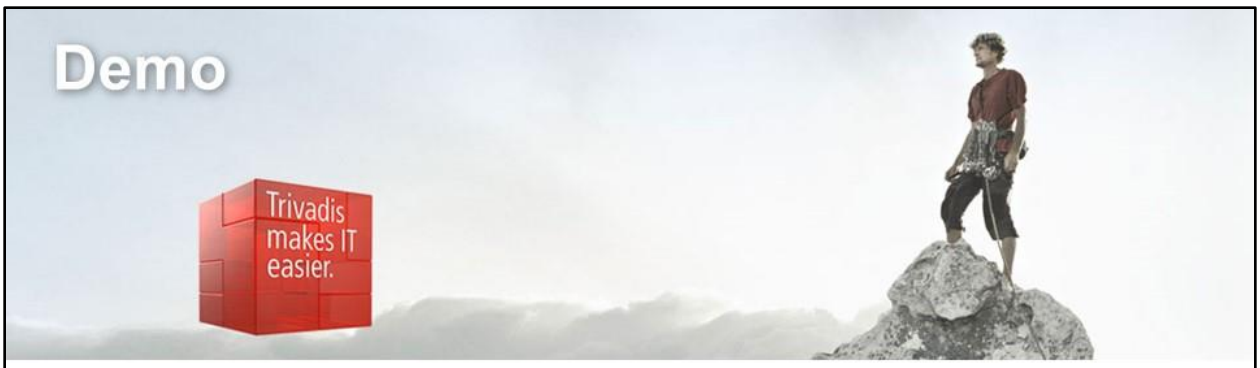
- Expose only the readonly observable

```
private selectedEmployeeSource = new Subject<Employee>();
selectedEmployeeChanges$ =
                     this.selectedEmployeeSource.asObservable();
```


Use the Subject

- Now we can subscribe to the observable

```
this.employeeService.selectedEmployeeChanges$.subscribe(
   selectedEmpl => this.employee = selectedEmpl;
);
```



trivadis
makes IT easier.

BehaviorSubject

- When subscribing to a BehaviorSubject, you get the last value of the chain
- Everything else is exactly as with a regular Subject

```
private selectedEmpSource = new BehaviorSubject<Employee>(null);
selectedEmployeeChanges$ =
                    this.selectedEmpSource.asObservable();
```

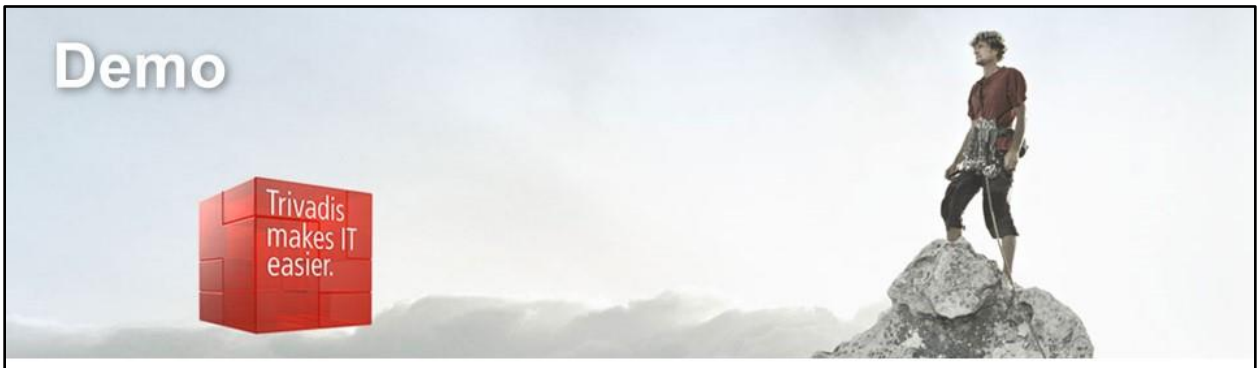

trivadis
makes IT easier.

Cleaning up Subscriptions

- To prevent memory leaks, we should unsubscribe from the observable if we are finished using it.

```
private sub: Subscription;

ngOnInit() {
  this.sub = this.selectedEmployeeChanges$.subscribe(empl =>
     (this.employees = empl)
  );
}
ngOnDestroy(): void {
  this.sub.unsubscribe();
}
```

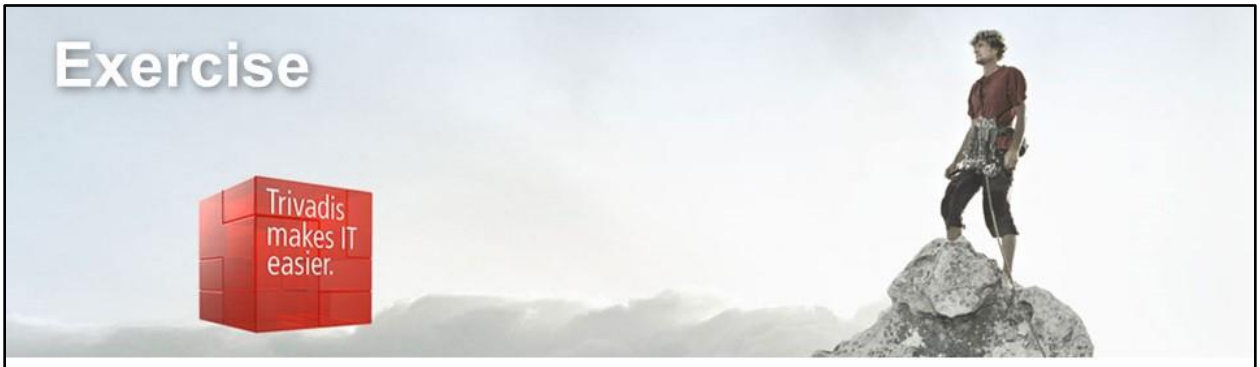

trivadis makes IT easier.



trivadis makes IT easier.

Summary: State Management with Notifications

- Use Subject when notifications are more than just changes to bound properties
- It does not need a subject if notifications are not required
- Especially not if notifications are only for changes to bound properties
- Use Subject if there is no initial value

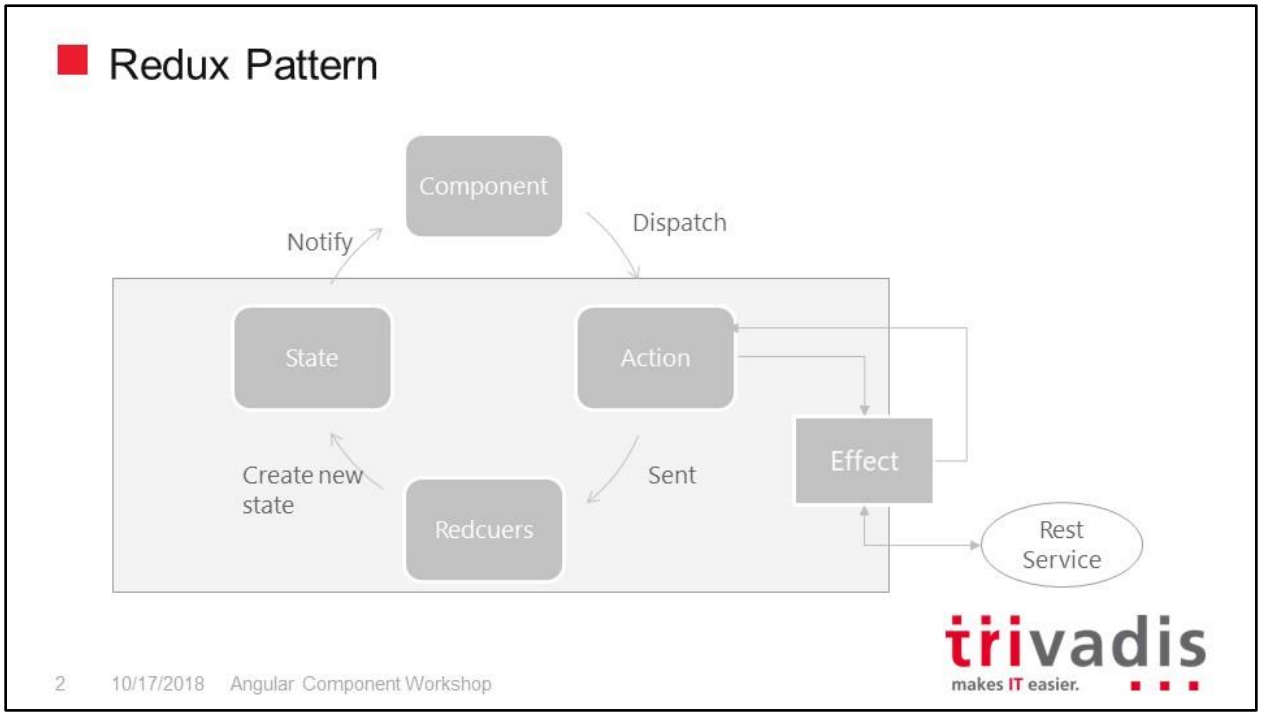


Add a ParamService for the employee-list.component.ts (Property Bag)

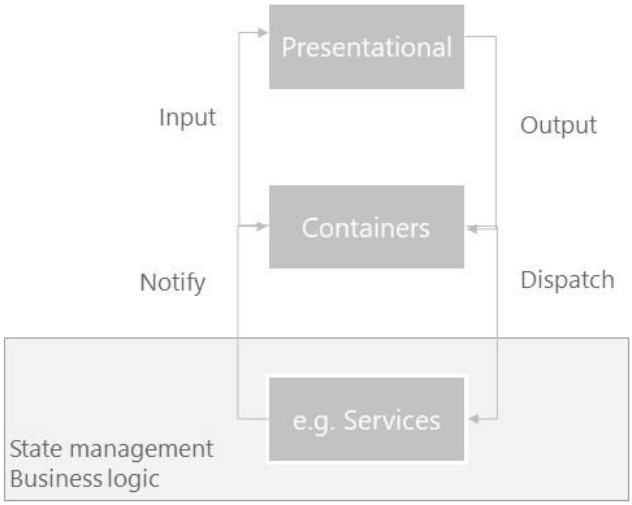
Add a basic state management service and use it with employee-shell.comoponent.ts

Change it to use a Subject or BehaviorSubject

06_Component Architecture

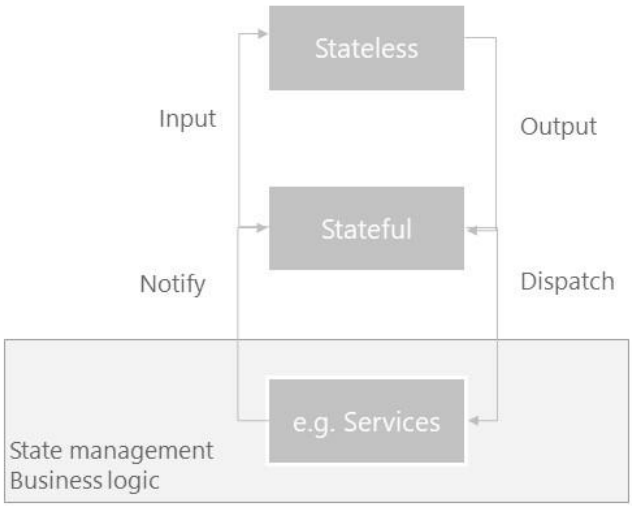


Component Architecture



trivadis
makes IT easier.

Component Architecture



trivadis
makes IT easier.

Advanced component architecture

Containers

- State-full Components
- Aware of State
- Read data from state/store
- Dispatch Actions

Presentational

- State-less Components
- Not aware of State
- Read data from @Input
- Callbacks via @Output

trivadis makes IT easier.

ChangeDetectionStrategy.OnPush

- OnPush means that the change detector's mode will be initially set to CheckOnce
- Angular only perform ChangeDetection if the reference of the input changes
- For our presentational components we can set ChangeDetectionStrategy to OnPush
  Because we fetch our data through an @Input
- For our container components we can also set ChangeDetectionStrategy to OnPush **if** we load everthing via an Observable and async pipe.

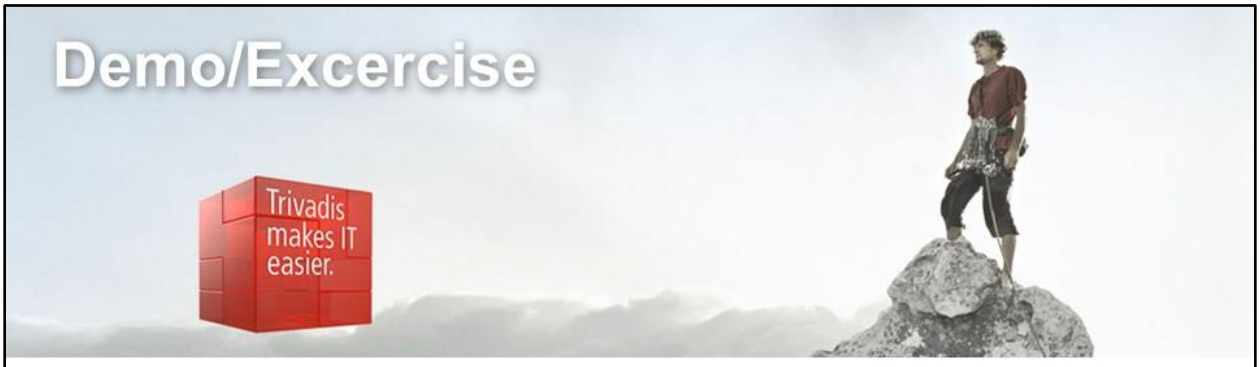
trivadis makes IT easier.

ChangeDetectionStrategy.OnPush

- Inside our component decorator

```
@Component({
    selector: 'app-employee-form',
    ...
    changeDetection: ChangeDetectionStrategy.OnPush
})
```


trivadis makes IT easier.

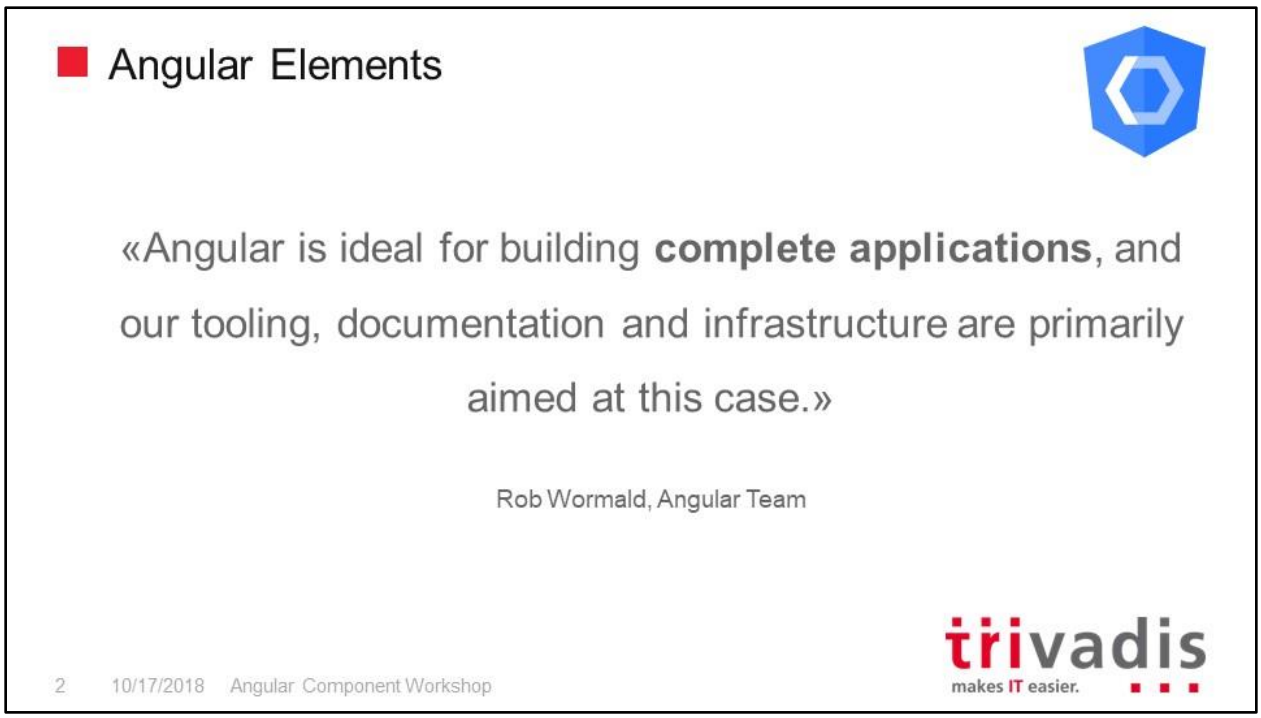


Check the concept of the employee-form

Add OnPush to the specific components

trivadis makes IT easier.

07_Angular Elements



Overview

trivadis makes IT easier.

Angular Elements

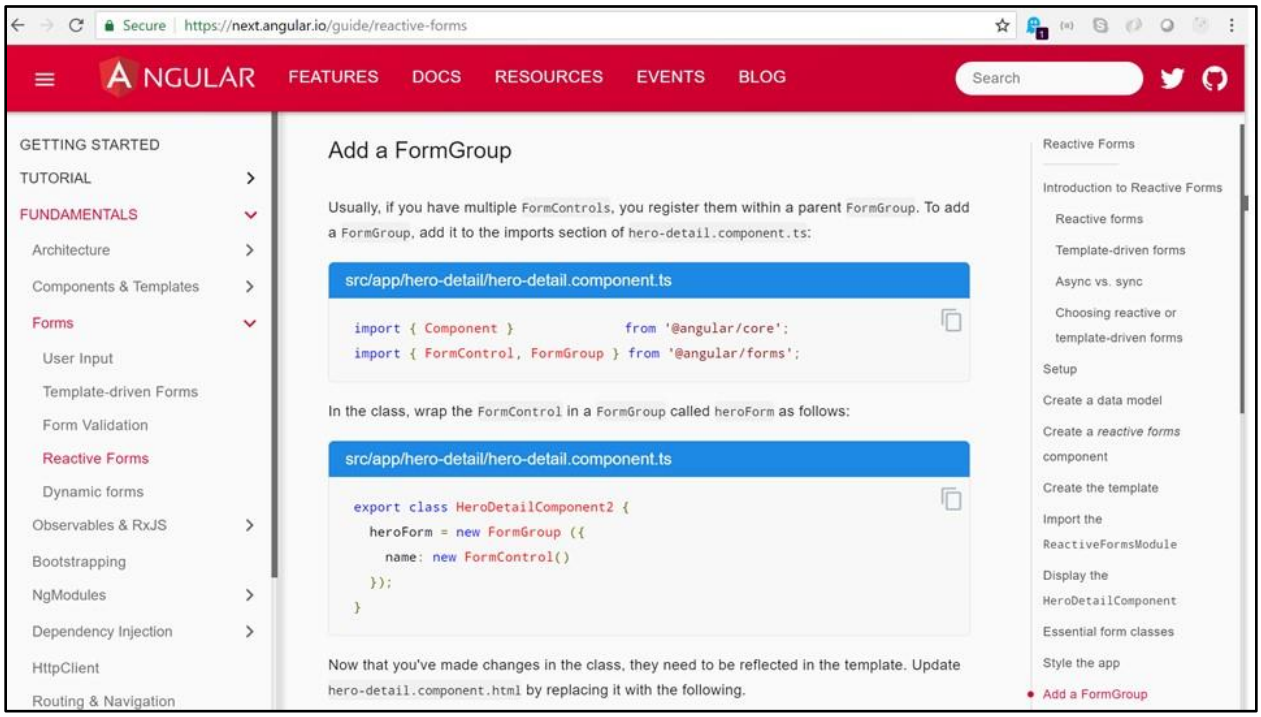
«[…] but it's quite challenging to use in scenarios that don't fit that specific Singe Page Application model.»

Rob Wormald, Angular Team

trivadis makes IT easier.

Use Cases

- Enhancing existing HTML Pages
- Content Management Systems
- Use components in other environments or frameworks
- Microfrontends
- Reuse components across teams



Web Components

Web Components are a set of features added by the W3C

- **HTML Template**: Template of the HTML
- **Shadow DOM**: DOM and style encapsulation
- **HTML Imports**: Imports in HTML
- **Custom Elements**: Ability to add to the HTML vocabulary

trivadis makes IT easier.

Custom Elements

Custom elements share the same API surface as native DOM objects:

- **Attributes**
- **Properties**
- **Methods**
- **Events**

trivadis makes IT easier.

Custom Elements

- Create and define a Custom Element
- Custom elements are **HTMLUnknownElement** until upgraded

```
class myElement extends HTMLElement {
        …
}

customElements.define('my-element', myElement);
```


Custom Elements: Reactions

```
class myElement extends HTMLElement {

  connectedCallback() {
    ...
  }
  disconnectedCallback() {
    ...
  }

}
```

Custom Elements: Attributes

```
class myElement extends HTMLElement {
 static get observedAttributes() {
    return ['country'];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'country') {
      // do something with newValue
    }
  }
}
```

- Can be used as follows:

```
<app-matches-by-country country="sui"></app-matches-by-country>
```


Custom Elements: Properties

```
class myElement extends HTMLElement {
 get country() {
    return this.getAttribute('country');
  }

  set country(val) {
    this.setAttribute('country', val);
  }
}
```

- Can be used as follows:

```
let matches = document.querySelector('app-matches-by-country');
matches.country = 'ger';
```

Custom Elements: Custom Events

```
class myElement extends HTMLElement {

 emitCountryChange() {
    this.dispatchEvent(
           new CustomEvent('country-change', {
             detail: this.country
        }));
  }
}
```

- Can be used as follows:

```
let matches = document.querySelector('app-matches-by-country');
matches.addEventListener('country-change', event => { ... });
```


Custom Elements in Angular

- Angular has been designed for this

```
<app-matches-by-country
       [country]="sui"
       (countryChanged)="foobar($event)"
       >
</app-matches-by-country>
```

Enter Angular Elements

- Provides a bridge from angular concepts to web components.

@HostBinding()	=>	Attributes
@Input()	=>	Properties
@Output()	=>	CustomEvents
Lifecycle Hooks	=>	Reactions

Create our first Angular Element

- Update Angular CLI (> Version 6).
- Create new project
- ng add @angular/elements
- ng g c matchesByCountry -v Native

Update Polyfill

- Add useful polyfill
  npm install @webcomponents/custom-elements --save
- Add polyfill to polyfill.ts

```
/* CUSTOM ELEMENTS */
// Used for browsers with partially native support of CE
import '@webcomponents/custom-elements/src/native-shim';

// Used for browsers without a native support of Custom Elements
import '@webcomponents/custom-elements/custom-elements.min';
```


Update Module

- Add Component to entryComponents
- Remove bootstrap Array.

```
@NgModule({
  imports: [BrowserModule, BrowserAnimationsModule,
              HttpClientModule, MatCardModule],
  declarations: [AppComponent, MatchesByCountryComponent],
  providers: [],
  entryComponents: [MatchesByCountryComponent]
})
export class AppModule {
       …
}
```

Update Module

- Add ngDoBootrap Method to body of class. This means to start the module but not render any component

```
@NgModule({ … })
export class AppModule {
    constructor(private injector: Injector) {}

   ngDoBootstrap () {
      const el1 = createCustomElement(MatchesByCountryComponent,
                  { injector: this.injector });
    customElements.define('app-matches-by-country', el1);
  }
}
```


Use Angular Element

- Add new HTML Tag to index.html

```
…
<body>
  <app-matches-by-country country="SUI"></app-matches-by-country>
</body>
…
```

- Run the application and see your Angular Element Component in action

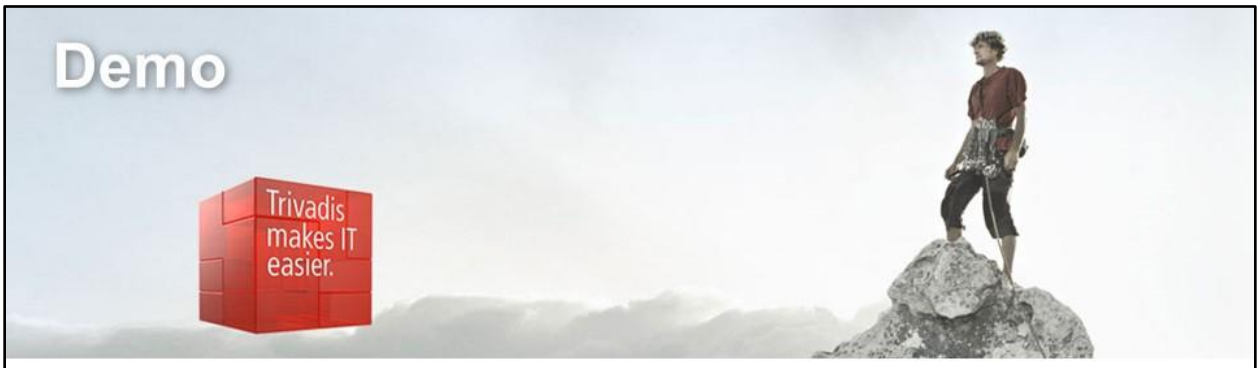
Use Angular Element

- Add HTML Tag by Javascript

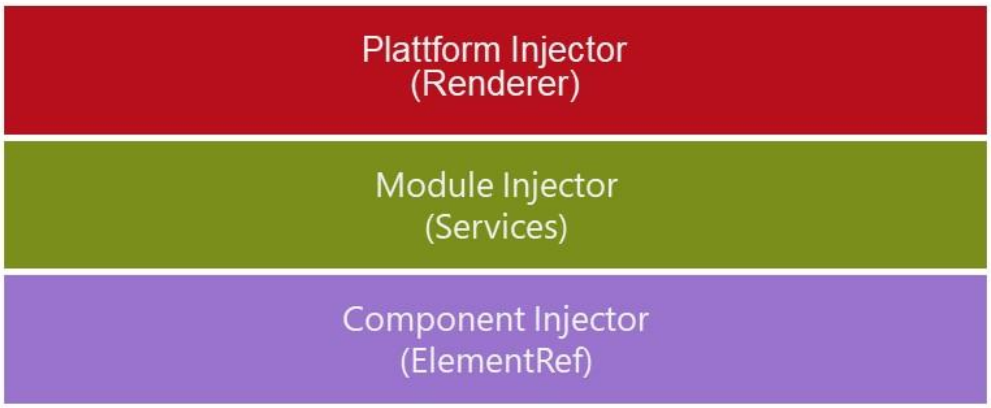
```
const matchesToday = document.createElement('app-matches-today');
document.body.appendChild(matchesToday);
matchesToday.country = 'SUI';
matchesToday.setAttribute('country', 'GER');
```

- Or by Custom Element API

```
const MatchToday = document.get('app-matches-today');
const matchToday = new MatchToday();
// or
const matchToday = new MatchToday(differencInjector);
```

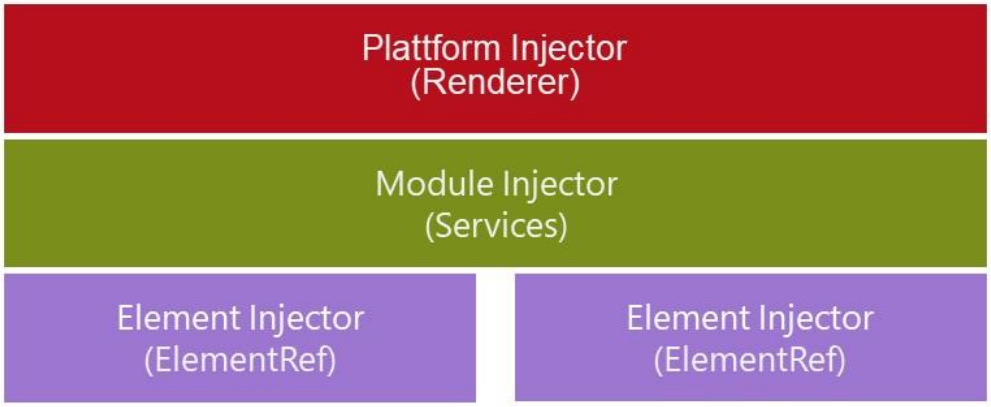



Dependency Injection



trivadis makes IT easier.

Dependency Injection

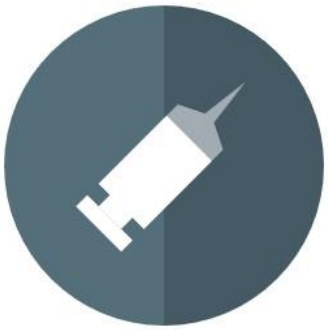


trivadis makes IT easier.

Module

```
@NgModule({ … })
export class AppModule {
    constructor(private injector: Injector) {}

   ngDoBootstrap () {
      const el1 = createCustomElement(MatchesByCountryComponent,
                    { injector: this.injector });
    customElements.define('app-matches-by-country', el1);
  }
}
```


Dependency Injection Works



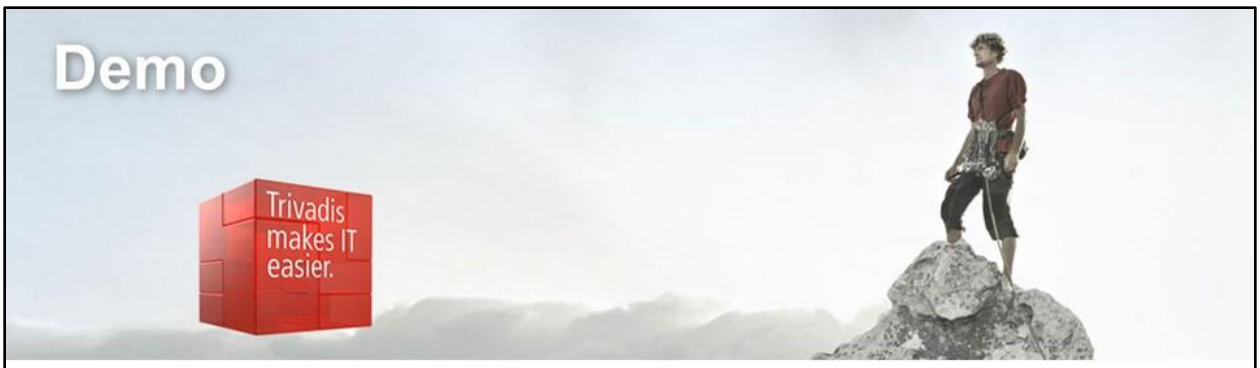
Content Projection

```
<app-test>
       <span>Hallo Angular Workshop</span>
</app-test>


@Component({
   selector: 'app-test',
   template: `
      <ng-content></ng-content>
   `
})
export class TestComponent {
}
```


Shadow DOM

- There are three ways of encapsulation
  - Emulated
    Styling from page has impact to component but not vice versa
  - Native / ShadowDom
    Styling from page has no impact to component
  - None
    Styling from component can impact the styling from the page

```
@Component({ …
       encapsulation: ViewEncapsulation.ShadowDom
})
export class MatchesTodayComponent implements OnInit { … }
```



trivadis
makes IT easier.

Build your component and pack it

- Install **concat** and **fs-extra**
- Add script command and build-elements.js

```
"build:elements": "ng build --prod --output-hashing none && node build-elements.js"
```

- Run the following command

```
npm run build:elements
```

- Open demo/index.html to see it in action

trivadis
makes IT easier.

build-elements.js

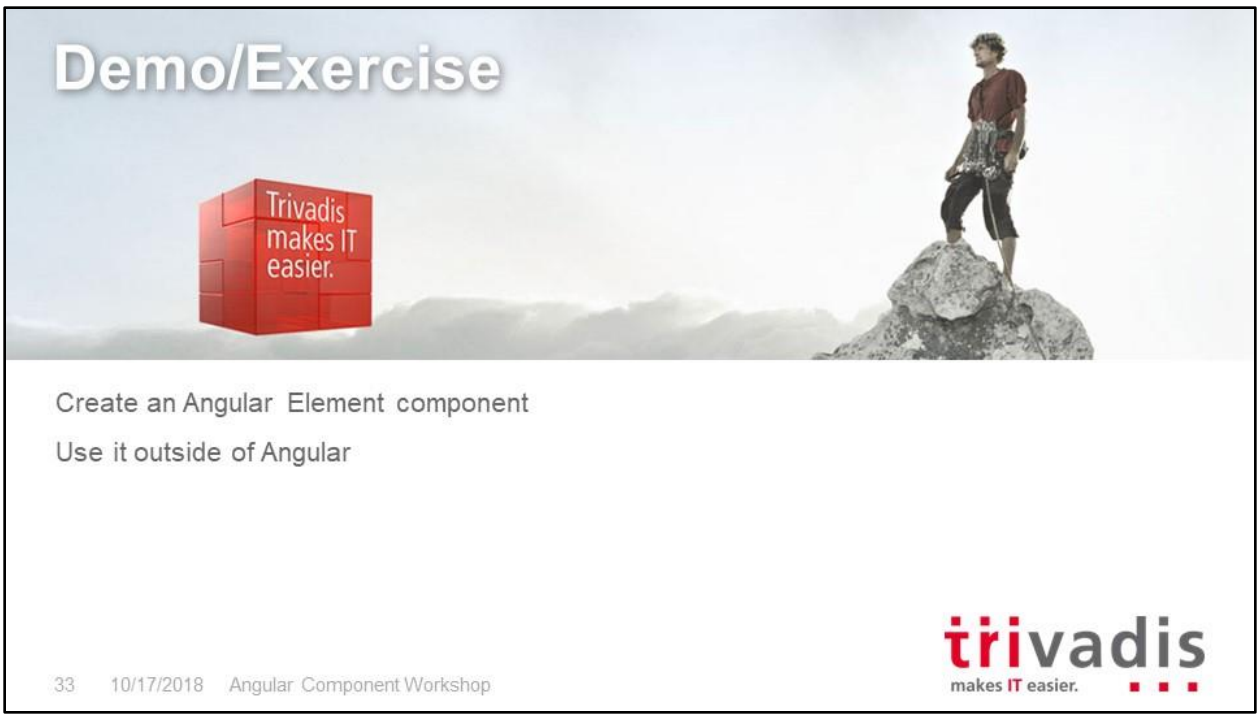
```
const fs = require('fs-extra');
const concat = require('concat');

(async function build() {
        const files = [
            './dist/WorldCupComponent/runtime.js',
            './dist/WorldCupComponent/polyfills.js',
            './dist/WorldCupComponent/main.js'
        ];

        await fs.ensureDir('elements');
        await concat(files, 'elements/world-cup.js');
        console.info('Wold Cup Element created successfully!');
})();
```


trivadis makes IT easier.

Angular Elements in V6

- It is just the beginning
- Size is too big for shipping in non Angular projects
- Will be much better with Ivy (V7)
- Will be much easier with V7
- Browser Support.

trivadis makes IT easier.



Create an Angular Element component

Use it outside of Angular

trivadis
makes IT easier.

08_Course Outro

Useful resources

- https://m.trivadis.com/angular
- https://github.com/AngularAtTrivadis/AngularComponentWorkshop
- https://marketplace.visualstudio.com/items?itemName=trivadis.ngtvd-extensions
- http://thomasgassmann.net/
- https://swissangular.com/

trivadis
makes IT easier.

Trivadis Angular Training

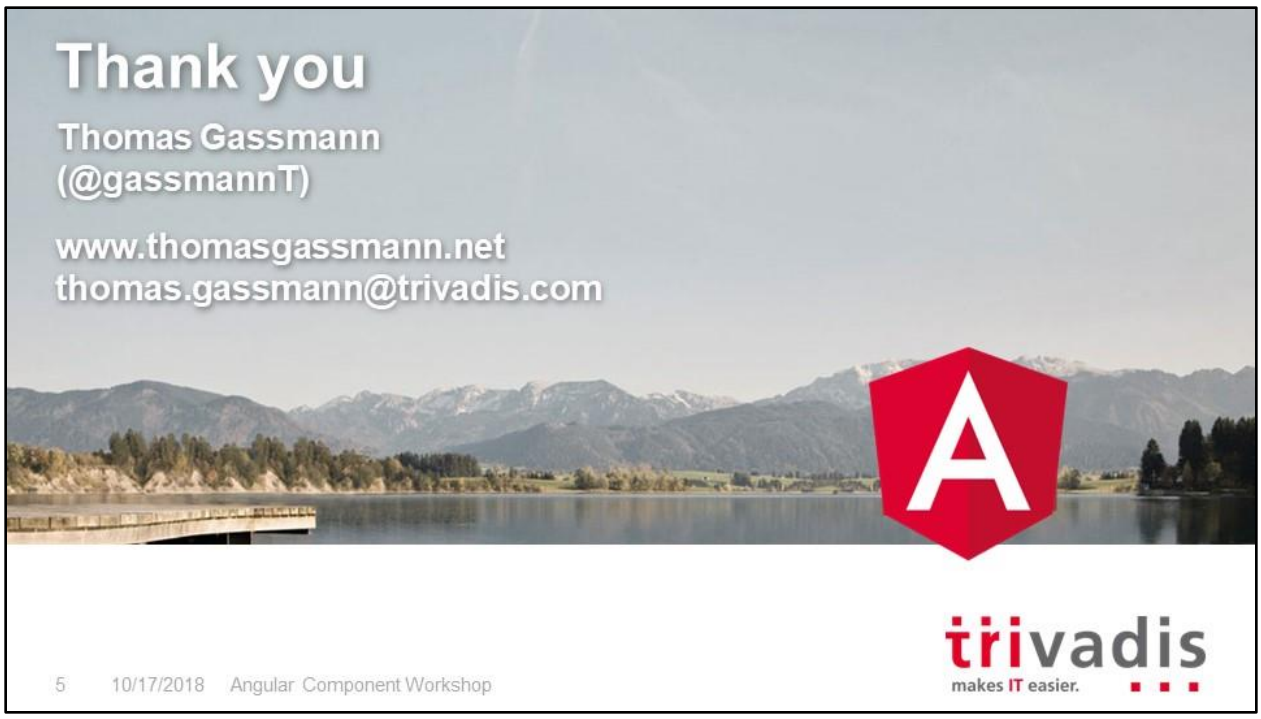
trivadis
makes IT easier.

Feedback

We are very grateful for your feedback.

- What was good?
- What could be improved?
- Something else?

trivadis
makes IT easier.


Thank you
Thomas Gassmann
(@gassmannT)
www.thomasgassmann.net
thomas.gassmann@trivadis.com
5 10/17/2018 Angular Component Workshop
trivadis
makes IT easier.