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

Angular Crash Course

Vers. 1.0.0 / September 2018

Autoren:
Thomas Huber, Thomas Gassmann, Thomas Bandixen

Course Content

00_Course_Introduction

Why TypeScript and Angular

- TypeScript brings the power of typed languages like C# / Java to the web-world
  - great to build mid- and large-sized applications
- Angular is a powerful Single Page Application (SPA) framework
  - Invented by Google
  - built with TypeScript
- With Web-technologies you can build apps for web, mobile and desktop

trivadis
makes IT easier.

Your Trainer

- 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 – 16:30
- If you need a break in between, feel free to ask
- Lunch today: Who is going to join?

trivadis makes IT easier.

Course Planning

- **Day 1: TypeScript and introduction to Angular**
  - Introduction to TypeScript
  - var and let
  - Interfaces and Classes
  - Functions
  - Generics
  - Modules and Namespaces
  - Introduction to Angular
  - Setting up an Application
  - Data Binding

Course Planning

- **Day 2: Angular**
  - Directives
  - Pipes
  - Modules
  - Routing
  - Http
  - Template-Driven Forms

Course Planning

- **Day 3: Angular**
  - Reactive-Forms
  - Testing
  - Redux
  - Deployment
  - Angular Elements
  - What is new in Version …

Introduce yourself

- name and function
- experience with Angular / TypeScript / JavaScript
- expectations for this course

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/AngularCrashCourse

Your machine

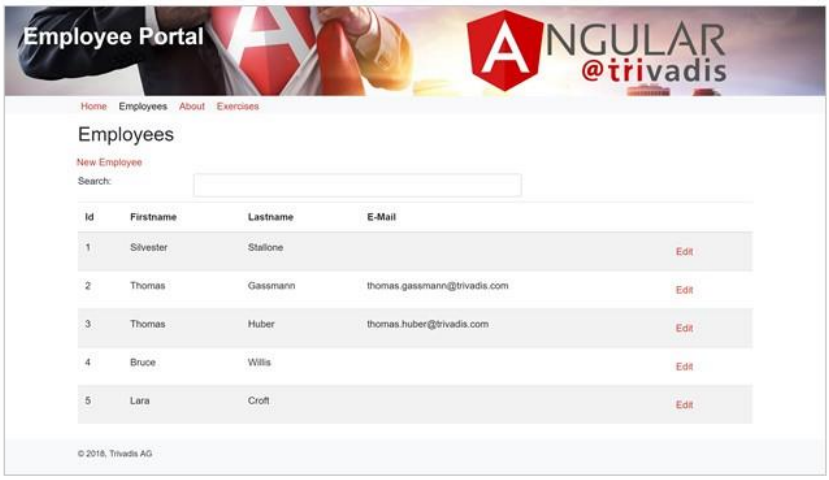
- Install Angular CLI

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

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

What are we building?

- A simple complete Angular project



trivadis makes IT easier.

Let's get our hands dirty!

trivadis makes IT easier.

01_TypeScript_Introduction

JavaScript popularity is exploding

- Great engines in browsers like V8
- Server-side with Node.js
- Package management with NPM (Node Package Manager)

JavaScript popularity is exploding

- Fantastic Frameworks: React, AngularJS, …
- Cross-platform with Electron and Cordova
- Great open source community

What's the problem with JavaScript?

The problem with JavaScript

- JavaScript was never engineered to write large applications
  - The language was written in 3 weeks
- It was intended to write apps with maybe 1'000 lines of code
  - Now people write apps with 100'000 or even 1'000'000 lines of code
- Writing large code bases can be hard with a dynamic language

- JavaScript does not have a static type system
  - No statement completion
  - No «Go to definition»
  - No «Find all references»
  - No «Refactorings»
  - No compile-time errors
  - …

trivadis makes IT easier.

TypeScript: JavaScript that scales

Makes it easier to build medium and large applications

trivadis makes IT easier.

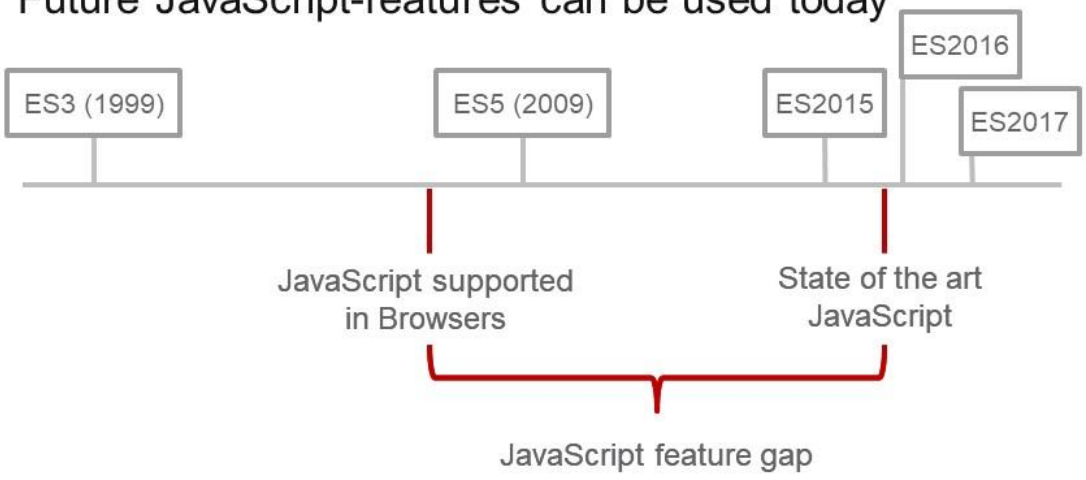
What is TypeScript?

- A statically typed superset of JavaScript
  - It «compiles» to plain JavaScript
- Works on any browser, any host, any OS
- Entire project is open source
  - https://github.com/Microsoft/TypeScript

What are the advantages of TypeScript

- Great tooling due to static types
  - Intellisense
  - Go to definition
  - …
- Future JavaScript-features can be used today
  - TypeScript allows you to use new features and compiles them to older JavaScript-versions

Future JavaScript-features can be used today



In 2016

trivadis makes IT easier.

Setting up TypeScript

- Install node and npm: www.nodejs.org
- Use npm for to install typescript

```
npm install -g typescript
```

- Instead of .js, use the .ts extension
  - TypeScript is a superset of JavaScript, so every JavaScript-code is valid TypeScript-code

trivadis makes IT easier.

Compile TypeScript

- Compile a .ts-file to a .js-file with the TypeScript-Compiler (tsc)

```
tsc main.ts
```

- Compile whenever the .ts-file changes (-w stands for –watch)

```
tsc -w main.ts
```


Tsconfig.json

- tsc looks for a tsconfig.json-file to read compiler-options
- Create such a file with

```
tsc -init
```

- In tsconfig.json, you can for example adjust the generated EcmaScript-Version:
  - es3, es5 or es6 / es2015

```
"compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false
    }
```

Tsconfig.json

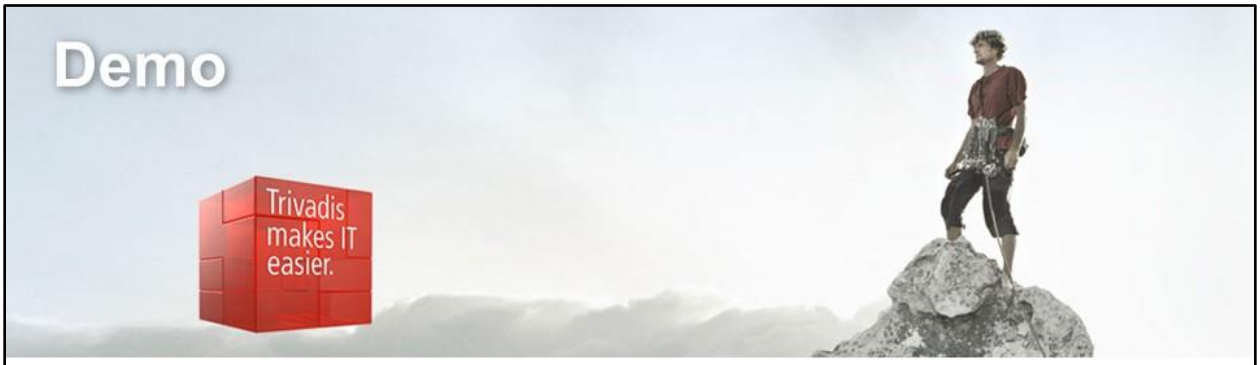
- If you have a tsconfig.json, compile just with

```
tsc
```

- Or start a watch-mode on your files with

```
tsc -w
```

- Take a look at the compiler-options you can set in tsconfig.json:
  - https://www.typescriptlang.org/docs/handbook/compiler-options.html



01_TypeScript_Introduction\demo01

Summary

- TypeScript is a statically typed superset of JavaScript
  – Any JavaScript is valid TypeScript
- TypeScript transpiles to ES3, ES5 or ES6
- Tooling support is great

02_TypeScript_BasicTypes

Overview

1. number
2. boolean
3. string
4. arrays
5. tuples
6. enums
7. void and never
8. undefined and null

trivadis
makes IT easier.

Type Inference

TypeScript finds out the type based on the assignment

```
let isVisible = false;
isVisible = "x" // Error: isVisible is boolean
```

But sure you can be explicit with a Type Annotation

```
let isVisible:boolean = false;
isVisible = "x" // Error: isVisible is boolean
```


Basic Types: boolean and number

```
let isVisible:boolean = false;
```

```
let height:number = 2;
height = 1.8;

height = "x"; // Error: type string is not assignable to number
```

number also supports hex, binary and octal literals

```
let dec:number = 27;
let hex:number = 0x001b;
let binary:number = 0b11011;
let octal:number= 0o0033;
```

Strings

```
let name:string ="Thomas";
name = 'Tom'; // like in JS single-quotes work as well
```

- Create a template-string by using back-ticks (``)
  - Use expressions with *${ exp }*

```
let welcomeMessage:string = `Hello ${name}, how are you?`;
```


Template strings

- The expression is not just a variable

```
let currentPage:number =1;
let nextPageMessage:string=`Go to page ${currentPage+1}`;
```

- A template string with back-ticks may contain linebreaks

```
let welcomeMessage:string = `Hello ${name}!
How are you?`;
```

The code above is equal to this statement:

```
let welcomeMessage = "Hello " + name + "!\nHow are you?";
```

Arrays

- There are two ways how to declare an array in TypeScript
  - Use the [] behind the type

```
let names:string[] =["Thomas","Sara","Julia"];
```

  - Use the generic Array-type

```
let names:Array<string> = ["Thomas","Sara","Julia"];
```


Iterating Arrays: for... of vs. for...in

```
let names:string[] =["Thomas","Sara","Julia"];
```

- **for... of** returns the values

```
for(let name of names){ console.log(name); }
// Output: Thomas, Sara and Julia
```

- **for ... in** returns the keys / indexes

```
for(let name in names){ console.log(name); }
// Output: 0, 1, 2
```

- Note: **for ... in** works not only on arrays, but on any object:
  - Great to iterate properties

Tuples

- Array with a fixed number of items
- Type of items can be different

```
let nameIsDev:[string,boolean] = ["Thomas",false];

nameIsDev[1] = true;  // OK
nameIsDev[1] = "yes"; // Error: string not assignable to boolean
```

- Next items are union types

```
let nameIsDev:[string,boolean] = ["Thomas",true];

nameIsDev[2] = "yes"; // OK
nameIsDev[2] = true;  // OK
```


Enums

- Give names to a set of numbers

```
enum Dock{Left, Top, Right, Bottom};

let dock:Dock = Dock.Left;
```

- Enums start with zero, but you can change it

```
enum Dock{Left = 1, Top, Right, Bottom};
```

- Or even number all elements

```
enum Dock{Left = 1, Top = 2, Right = 4, Bottom = 8};
```

Enums

```
enum Dock{Left = 1, Top, Right, Bottom};
let dock:Dock = Dock.Left;

// enum to number (val will be 1)
let val:number = dock;

// enum to string (name will be "Left")
let name:string = Dock[dock]; // or Dock[1];

// number to enum
dock = 2;

// string to enum
dock = Dock["Top"];
```


The «any»-type

- Sometimes you don't want type-checking
  - For example when you have dynamic content

```
let age = 20; // age is now a number due to the assignment

age = "sooo young"; // error: string not assignable to number
```

- Use the any-type to opt-out of type-checking

```
let age:any = 20; // use the any-type

age = "sooo young"; // OK, now a string is assignable
```

The «any»-type

- Allows you to call methods not known at compile-time

```
let age:any = 20;
var isYoung:boolean =     // OK at compile-time, but the method
  age.IsThisYoung(); // does not exist on the number 20
```

- Tooling doesn't know anymore about specific members

```
let name:any = "Thomas";

let length:number = name.length; // No intellisense for "length"
```


Type assertions

Type assertion: like a casting, but only for the compiler! It has no runtime impact!

```
let name:any = "Thomas";
let length:number = name.length; // No intellisense for "length"
```

- Use the as-syntax

```
let length:number = (name as string).length;
```

- Use the <>-syntax

```
let length:number = (<string>name).length;
```

- Note: Which one you use is personal preference
  - <>-syntax does not work with JSX

Type checks with typeof (classical JavaScript)

```
function append(text: string, appendix: any): string {
   if(typeof appendix === "number")
   {
       return text + Array(appendix).join(" ");
   }
   if(typeof appendix === "string")
   {
       return text + appendix;
   }
   throw new Error("appendix must be string or number);
 }
```

```
append(Thomas, true); // runtime error
```


Union Types

```
function append(text:string,appendix:string|number):string
 {
   if(typeof appendix === "number")
   {
       return text + Array(appendix).join(" ");
   }
   if(typeof appendix === "string")
   {
       return text + appendix;
   }
   // throw new Error("appendix must be string or number);
 }
```

```
append(Thomas, true); // compile-time error, as true
                      // is neither string nor number
```

Union types

- Prefer union types over any if possible

```
let age:string|number = 20;

age = "sooo young"; // OK

age = true; // Error, boolean is neither string nor number
```


Void and Never

- Void is used for functions don't return a type

```
function addLine(message:string):void
{
    document.write(message+"<br>");
}
```

- Never is used for functions that have an unreachable endpoint
  - Either endless loop or always throwing an error

```
function writeOutput():never
{
    throw Error("Don't use this")
}
```

Undefined and null

- undefined and null have their own types: undefined & null
- By default undefined/null are subtypes of all other types
  - Which means you can set for example a number or string to null or undefined

```
let name:string = "Thomas";
name = null; // OK
name = undefined; // OK
```

- undefined/null can lead to runtime-errors, as you might access members of a variable that is either null or undefined

Undefined and null: the evil part

```
function getNumber():number
{
    var x;
    return x; // x is undefined
}

getNumber().toString(); // Runtime-error: toString not
                        // available on undefined
```

Strict null-checking

- By default undefined/null are subtypes of all other types
- TypeScript 2.0 has a strict-null-checking option
- Turn it on, and undefined/null are no more subtypes of all other types
  - null and undefined are only assignable to themselves and any
  - If you want to have a nullable string, you need to use a union type
  - This is great to avoid errors like on the previous slide

Strict null-checking

- Turn it on in the compiler-options in tsconfig.json

```
"compilerOptions": { ..."strictNullChecks": true    }
```

- With strict null-checks

```
let name:string = "Thomas";
name = null; // ERROR
name = undefined; // ERROR
```

- Use union-types if you want for example support null

```
let name:string|null = "Thomas";
name = null; // OK
name = undefined; // ERROR
```

Undefined and null: with strict null-checking

```
function getNumber():number
{
    var x;
    return x; // x is undefined,
              // => leads now to a compile-error
}
```

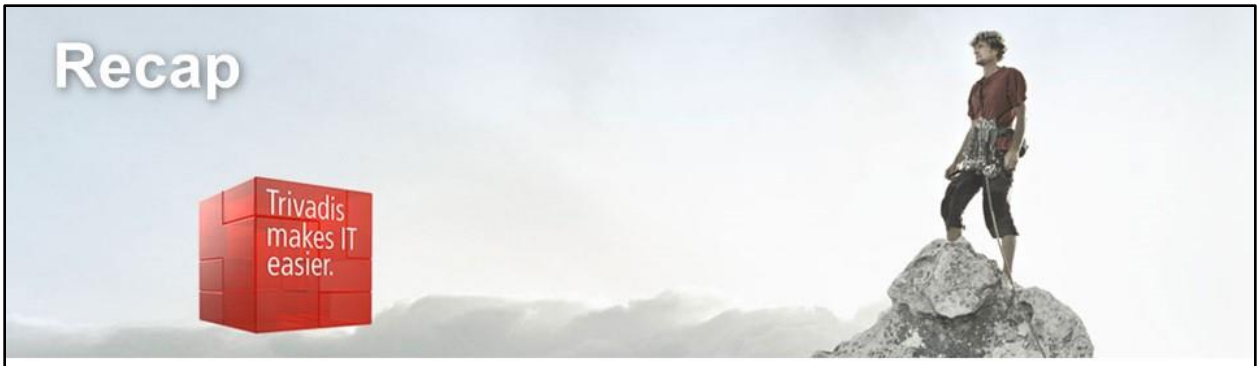
The method above leads to a compile error:

```
error TS2322: Type 'undefined' is not assignable to type 'number'.
```


Summary

- TypeScript has different basic types
  – number, string, boolean, array, tuple, enums
- Use any for dynamic data
- Use Union Types to combine types
- Turning on StrictNullChecks in the TypeScript-Compiler is recommended

Recap



demo01 – demo06

trivadis
makes IT easier.

03_TypeScript_VarAndLet

var, let and const

- *var* is the classical way to define a variable in JavaScript

```
var name = "Thomas";
```

- *let* and *const* are new types for variable declarations in JavaScript
  – As TypeScript is a superset, they're also there

```
let name: string = "Thomas";
```

- Prefer *let* and *const* over *var*!
  – Why? Let's look at *var* in detail

Var: scoping rules

- You can access var declarations anywhere within their containing function (like function parameters)
  - or within their containing module, namespace, or global scope

```
function getNumber(init)
{
    if(init)
    {
        var x = 9;
    }
    return x;
}
```


Var: scoping rules

- Changing var to let will produce a runtime error in JavaScript
- Changing var to let will produce a compile-time error in TypeScript

```
function getNumber(init)
{
    if(init)
    {
        let x = 9;
    }
    return x; // x is not visible here, as "let" is block-scoped
}
```

Var: scoping rules

Another one! What's the output here?

```
function writeToBody()
{
    for(var i = 0; i<5; i++)
    {
       setTimeout(function() {
           document.write(i+"<br>");
       }, 200);
    }
}
```

```
5
5
5
5
5
```

expected

```
0
1
2
3
4
```


trivadis makes IT easier.

Var: scoping rules

JavaScript workaround: Nest another function expression that is called immediately with a parameter

– the parameter is function-scoped and so captured

```
function writeToBody()
{
    for(var i = 0;i<5;i++)
    {
        (function (i)
        {
            setTimeout(function() {
                document.write(i+"<br>");
            }, 200);
        })(i);
    }
}
```

```
0
1
2
3
4
```


trivadis makes IT easier.

Var: scoping rules

JavaScript workaround: Nest another function expression that is called immediately with a parameter

– the parameter is function-scoped and so captured

```
function writeToBody()
{
    for(var i = 0;i<5;i++)
    {
        (function (val)
        {
            setTimeout(function() {
                document.write(val+"<br>");
            }, 200);
        })(i);
    }
}
```

```
0
1
2
3
4
```


Var: scoping rules

Workarounds are evil. ☺ Just use *let* instead

- this will create a new scope per iteration

```
function writeToBody()
{
    for(let i = 0; i<5; i++)
    {
        setTimeout(function() {
            document.write(i+"<br>");
        }, 200);
    }
}
```

```
0
1
2
3
4
```

let declarations

- Looks similar like a var declaration. Just a different keyword

```
let name: string = "Thomas";
```

- are block-scoped
- are pretty much like declarations in Java / C#

Let declarations

- You can't declare a variable twice in the same scope

```
let name: string = "Thomas";
let name: string = "Bill" // Error
```

- But you can shadow it in another scope

```
let name="Thomas";
{
   let name="Bill";
   console.log(name); // Bill
}
console.log(name); // Thomas
```

let declarations

- Variable must be declared before you access it

```
console.log(name); // Error: name is not defined
let name: string = "Thomas"
```

- Here «name» is accessed after it is declared, that's OK:

```
function printName()
{
    console.log(name);
}
let name = "Thomas"

printName(); // OK
```


Const declarations

- Declared and block-scoped like let

```
const carWheels: number = 4;
```

- Only assignable once:

```
const carWheels: number = 4;
carWheels = 3; // Error: Assignment to const variable
```

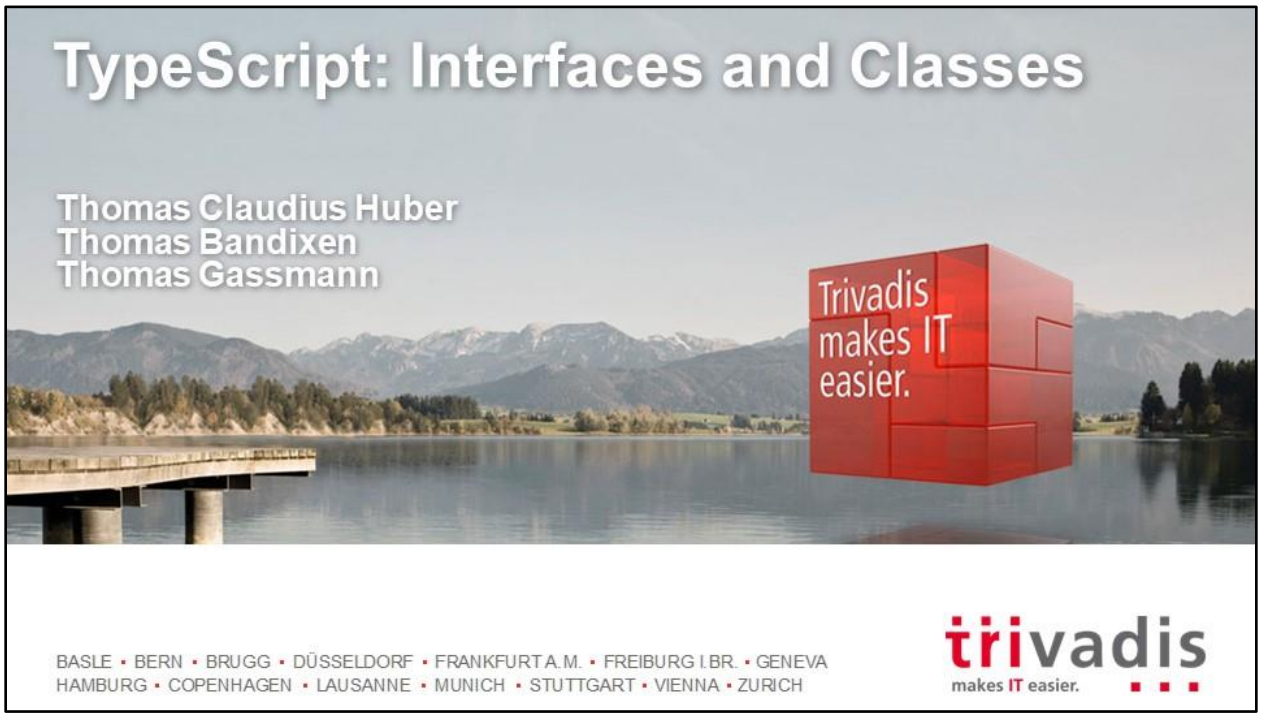
- Properties of complex types are still adjustable

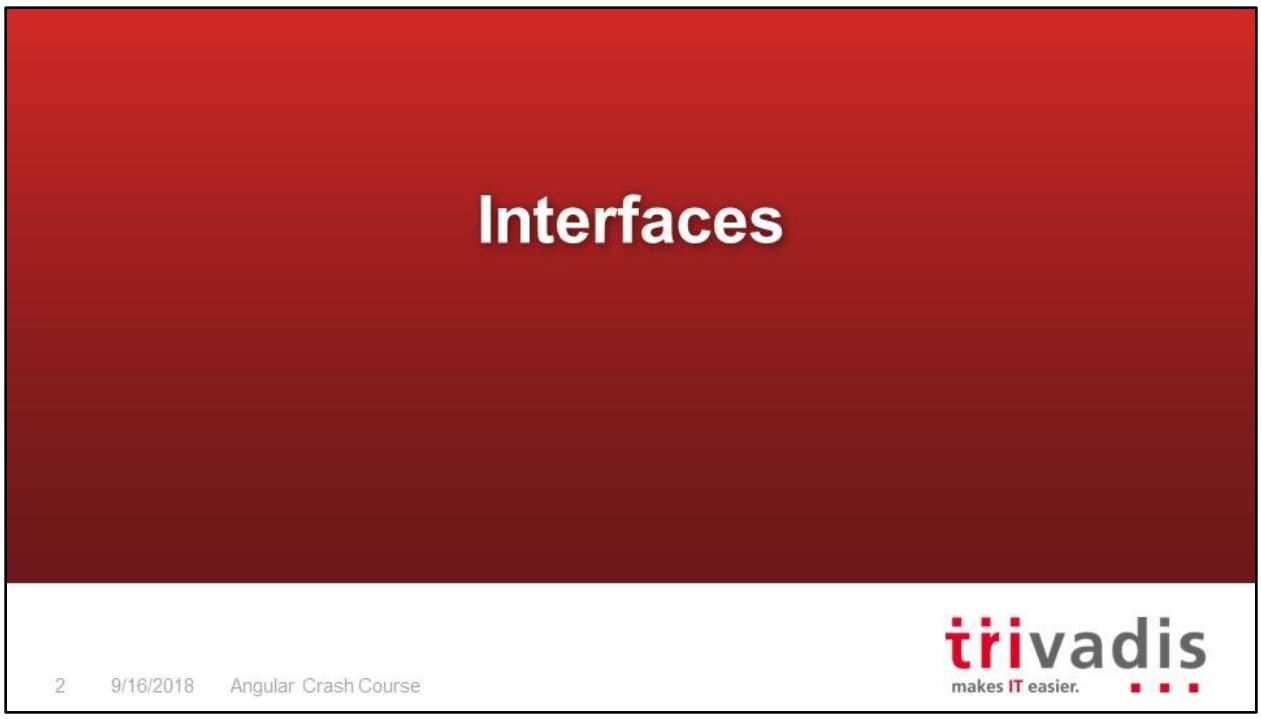
```
const car = {wheels: 4,type: "bmw"};
car.wheels = 3; // OK
car.type = "Trike"; // OK
car = {wheels: 2, type: "harley"} // ERROR: Assignment to
const...
```

Summary

- As TypeScript is a JavaScript-superset, let and const are available
- Prefer let and const over var

04_TypeScript_InterfacesAndClasses





Interfaces

- Interfaces are used to define contracts for your code
  - Even for other code that is using your code
- Makes it for example very clear what to pass to a function
- Allows you to create clear boundaries for your code

Without interfaces

```
function getFullName(friend)
{
    var fullName = friend.firstName;
    if(friend.lastName){ fullName += " " + friend.lastName; }
    return fullName;
}

var fullName = getFullName({firstName: "T", lastName: "Huber"});
console.log(fullName); // Logs "T Huber"

fullName = getFullName({firstName: "Thomas"});
console.log(fullName); // Logs "Thomas"

fullName = getFullName({firstname: "Thomas"});
console.log(fullName); // Logs "undefined"
```

Let's introduce an interface

- Use the interface keyword
- define optional properties with a **?** after the name
- An «I» prefix like in C# / Java is normally not used in TypeScript

```
interface Friend {
    firstName:string;
    lastName?:string;
}
function getFullName(friend: Friend) {
    ...
}
```


With the interface, a typo leads to an error

```
interface Friend {
    firstName: string;
    lastName?: string;
}
function getFullName(friend: Friend)
{
    ...
}

fullName = getFullName({ firstname: "Thomas" }); // Error
```

Interfaces with functions

- Beside properties, you can also define interfaces for functions
- Instead of defining the function type...

```
function printFriend(printer:(f: Friend) => void, friend: Friend)
{
    printer(friend);
}
```

- ... you can use an interface

```
interface FriendPrinter{
    (f: Friend): void
}
function printFriend(printer: FriendPrinter, friend: Friend)
```


Generated code for an interface

- Interfaces are just a TypeScript concept
- There's no JavaScript-code generated for an interface
- The interface is «just» important for compile-time checks!
  – And of course for tooling

Classes

Classes

- ECMAScript2015 (ES6) introduces classes
  – a concept object-oriented programmers are familiar with
- TypeScript allows you to use classes and compile them to a JavaScript-version that is available in all major browsers

Create Classes with constructors and methods

```
class Friend {
    firstName: string;
    lastName?: string;

    constructor(firstName: string, lastName?: string)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName(): string {
         return `${this.firstName} ${this.lastName}`;
    }
}
```


Instantiate a class

- To create an instance, call the constructor with **new**

```
let friend = new Friend("Thomas", "Huber");
```

- On your instance, access properties and methods

```
let friend = new Friend("Thomas", "Huber");
let firstName = friend.firstName;
let fullName = friend.getFullName();
```

Class-members can have modifiers (incl. constructor)

- **public** – visible from outside – **this is the default**
- **protected** – visible in subclasses
- **private** – only visible inside the class

Add Modifiers to Constructor parameters

- This will generate properties for the parameters
  – It's a shorthand syntax called **Parameter Properties**
- The whole Friend-class can be written like this:

```
class Friend {
    constructor(public firstName: string, public lastName?: string)
    {
    }

    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

Defining Properties with Accessors

- Sometimes you want to hook in when a property is set or get

```
class Friend {
    private _firstName: string;

    get firstName(): string { return this._firstName; }

    set firstName(value: string) { this._firstName = value; }

    constructor(firstName: string)
    {
        this._firstName = firstName;
    }
}
```


Defining Properties with Accessors

- Usage with Accessors is like without

```
let friend = new Friend("Thomas");

friend.firstName = "Angular";
let firstName = friend.firstName;
```

- But now you have the power to do something when the property is get or set

Static properties

- You can define static properties on a class with the **static** keyword

```
class Person
{
  static counter: number=0;

  constructor() {
   Person.counter++;
  }
}
```

```
let p1 = new Person();
let p2 = new Person();
let p3 = new Person();

// This will log «3»
console.log(Person.counter);
```


Readonly properties

- You can define readonly properties with the **readonly** keyword

```
class Person
{
  readonly name: string;

  constructor(name: string)
  {
    this.name = name;
  }
}
```

```
let p = new Person("Thomas");

let personName = p.name; // OK

p.name ="Angular"; // Error
```

Use inheritance

- You can build up a class hierarchy in TypeScript

```
class Animal {
    constructor(public name: string){}
}

class Frog extends Animal {
    jump(distance: number) {
        console.log(`the frog ${this.name}
is jumping ${distance}m`);
    }
}
```


Use inheritance

- Call base-constructors from subclasses via **super**

```
class Animal {
    constructor(public name: string){}
}

class Frog extends Animal {
    constructor(name: string, public distance: number)
    {
      super(name);
    }
    ...
}
```

Abstract classes

- Force a subclass to implement something for you
- Define the class with the **abstract** keyword
- Define abstract members also with the **abstract** keyword

```
abstract class Person {
    constructor(public name: string){}

    abstract sayHello(): void;
}
```


Extend an abstract class

- Just implement the abstract members and you're fine!

```
abstract class Person {
    constructor(public name: string){}
    abstract sayHello(): void;
}

class Friend extends Person {
  sayHello() {
    console.log(`Hello my friend ${this.name}`);
  }
}
```

Implementing interfaces

- Use the **implements** keyword

```
interface Animal {
   name: string;
}
class Frog implements Animal {
    constructor(public name: string, public distance: number)
    {
    }
    ...
}
```


Implementing multiple interfaces

- Just separate them with a comma

```
interface Animal {
   name: string;
}
interface Jumper {
   jump() => void;
}
```

```
class Frog implements Animal, Jumper {
   name: string;
   jump() {
       console.log("Flying high");
   }
}
```

Type Compability

trivadis makes IT easier.

Type Compability in TypeScript

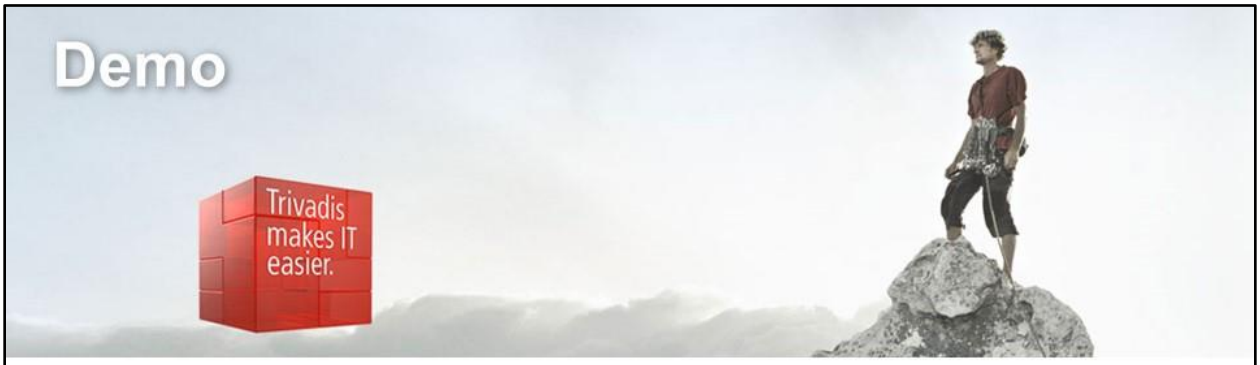
- structural instead of nominal typing
  - members are important, not the types themselves!

```
interface Developer {
    githubusername: string;
}

// Note: Person does not implement Developer-interface
class Person {
    githubusername: string;
}

let p: Developer = new Person(); // OK, because member exists
```


trivadis makes IT easier.

Summary

- TypeScript supports interfaces and classes like known from OO-languages like C# or Java
- Interfaces don't generate any JavaScript-code



04_TypeScript_ClassesAndInterfaces
Demo 1 – Demo 6

05_TypeScript_Functions

JavaScript supports named and anonymous functions

Named function

```
function multiply(x, y) {
    return x * y;
}
```

Anonymous function

```
let add = function(x, y) { return x + y; };
```

Call them

```
let resultMul = multiply(3, 3);
let resultAdd = add(3, 3);
```


trivadis
makes IT easier.

With TypeScript you can add types to your functions

- To ensure they correct types are passed in
- Define the parameter-types and the return-type

```
let add = function(x: number, y: number): number {return x + y;};
```

- You can even explicitly define the type of the variable

```
let add:(a: number, b: number) => number
  = function(x: number, y: number): number { return x + y; };
```


Define optional parameters

- You cannot omit parameters in TypeScript if they're
  – not optional and if there's no default value
- Make parameters optional with a **?**

```
function getFullName(firstName: string, lastName?: string) {
    if (lastName)
        return `${firstName} ${lastName}`;
    else
        return firstName;
}
console.log(getFullName("Thomas"));
console.log(getFullName("Thomas", "Huber"));
console.log(getFullName()); // Error: firstName parameter missing
```

Define default values for parameters

- Define default values for parameters to omit them

```
function getFullName(firstName: string = "Thomas", lastName?: string) {
    if (lastName)
        return `${firstName} ${lastName}`;
    else
        return firstName;
}


console.log(getFullName()); // OK, prints "Thomas"
```


Define rest parameters

- If you don't know the number of parameters, use ... and a name

```
function getFullName(firstName: string, ...moreNames:string[]) {
    return firstName + " "+ moreNames.join(" ");
}

console.log(getFullName("Thomas"));
console.log(getFullName("Thomas", "Claudius", "Huber"));
console.log(getFullName("Thomas", ...["Claudius", "Huber"]));
```

Functions and «this»

- What is this refering to in the code below?

```
class NameDisplayer {
   public loadedName: string

   Load(){
       loadData(function(name: string){ this.loadedName=name; });
   }
}

function loadData(callback: (name: string) => void) {
    callback("Thomas");
}
```


Functions and «this»

- loadedName is undefined, as this was captured for the function

```
var x = new NameDisplayer();
x.Load();
console.log(x.loadedName); // undefined
```

Arrow functions for the win

- What is this refering to in the code below?

```
class NameDisplayer {
   public loadedName: string

   Load(){
       loadData((name: string) => { this.loadedName=name; });
   }
}

function loadData(callback: (name: string) => void) {
    callback("Thomas");
}
```

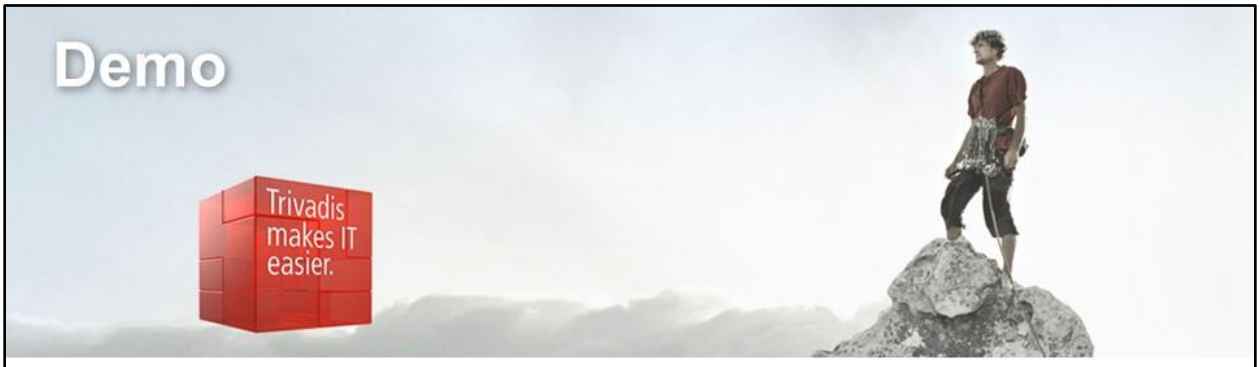

Functions and «this»

- loadedName is «Thomas», as this was captured from outside of the arrow function

```
var x = new NameDisplayer();
x.Load();
console.log(x.loadedName); // "Thomas"
```

Summary

- JavaScript is functional
- With TypeScript you can define
  - optional parameters
  - default values
  - rest parameters
- Use arrow functions if you want «this» to be the calling context



05_TypeScript_Functions
demo04

06_TypeScript_Generics

What are Generics

- Usually code that works with different types
- The types are defined via type-parameters
- Generics can be applied to functions, interfaces and classes

trivadis
makes IT easier.

TypeScripts Array class is generic

- Note how the generic type parameter is passed in **<>**
- The type parameter is part of the type
  – that means the type is **Array<number>** and not f.e. Array<any>

```
let numberArray = new Array<number>();


numberArray.push(1);
numberArray.push(2);
numberArray.push("Hello"); // Error, not a number
```


Generic Functions

- Allow you to use them with different types

```
function LogToConsole<T>(item: T): T {
  console.log(item);
  return item;
}
```

- Called with a string returns a string

```
let firstname: string = LogToConsole("Thomas");
```

- Called with a number returns a number

```
let luckyNumber: number = LogToConsole(13);
```

Generic Interfaces

- Add the Type-parameter to the Interface

```
interface IRepository<T> {
  Load(): T;
  Save(item: T): void;
}
```

```
class NumberRepo implements IRepository<number> {
  private _number=5;
  Load(): number { return this._number; }
  Save(item: number): void { this._number = item;}
}
```


Generic Classes

- Add the type-parameter also with <>

```
class GenericRepo<T> implements IRepository<T> {
  private _store: T;
  constructor(initialValue: T) {
   this._store=initialValue;
  }
  Load(): T {
    return this._store;
  }
  Save(item: T): void { this._store = item;}
}
```

Generic Classes

- Use your generic class

```
let repo = new GenericRepo<number>(5);

repo.Save(13);

repo.Save("Thomas"); // ERROR, as it's a number-repo
```


Generic constraints

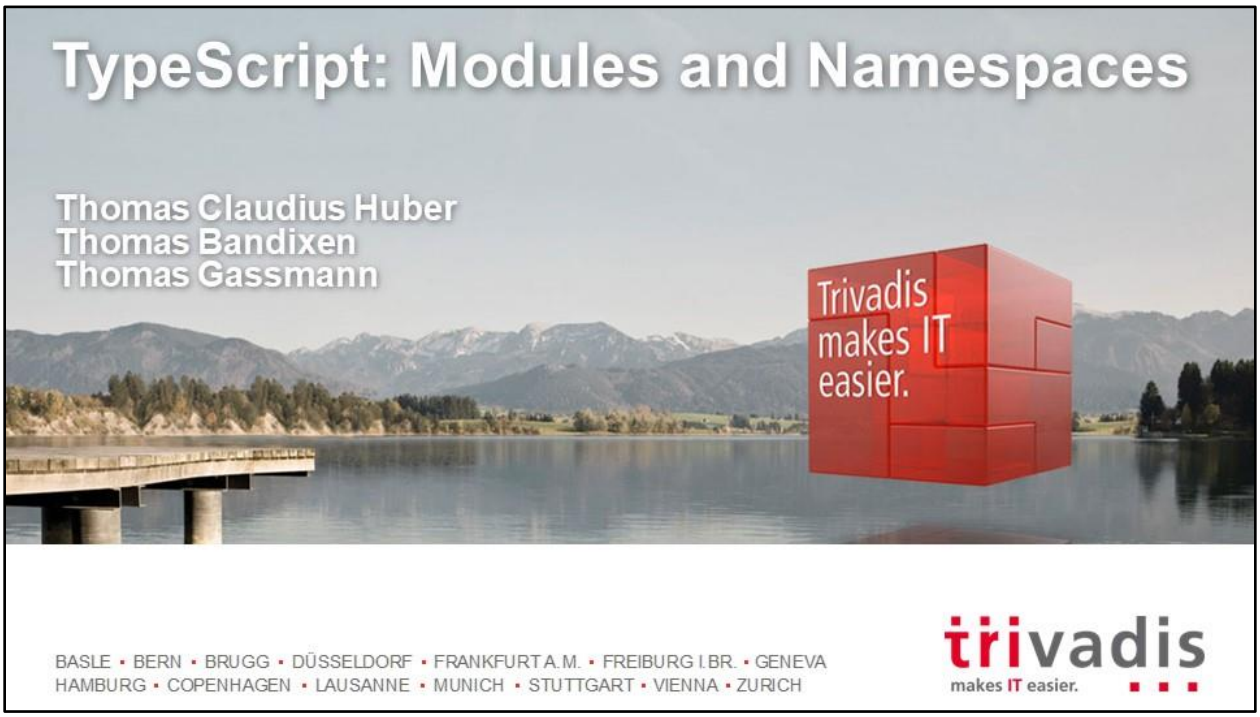
- Sometimes you want to limit the Type parameter

```
interface IName {
  name: string;
}

class GenericPrinter<T extends IName> {
  print(item: T) {
    console.log(item.name);
  }
}
```

Summary

- Generics are a powerful construct
- Great way to re-use code
- Works on
  - functions
  - interfaces
  - classes

07_TypeScript_ModulesAndNamespaces



What are Namespaces and Modules in TypeScript

Namespaces

- for code organization
- No module loader required
- ok for smaller apps

Modules

- for code organization
- supported natively in Node.js
- Browser needs a module loader
- TypeScript supports ECMA2015 module syntax

trivadis
makes IT easier.

Namespaces

```
namespace Persons {
   export interface Name{
       getFullName():string;
   }

    export class Person implements Name{
        constructor(public firstName: string,
                    public lastName: string) { }

        getFullName() {
            return this.firstName + " " + this.lastName;
        }
    }
}
```


Namespaces

- To access exported objects from a namespace, use the namespace-name followed by a dot

```
var p: Persons.Name = new Persons.Person("Thomas", "Huber");

console.log(p.getFullName());
```

Namespaces

- If you split your code across several files, just let TypeScript create one single output file
- This works fine with namespaces

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false,
        "outFile": "index.js"
    }
}
```


Modules

- Every serious dev is using modules instead of namespaces
  - native part of Node.js and ECMAScript2015
- Modules are based on the file and folder structure
  - a file that has exports is a module, that's it!
- In 2017 browsers don't natively support modules
  - you require a module loader like for example Webpack
  - just generating a single output-file like we did for namespaces is not enough for modules

Exporting from a Module

- A module is just a file
- Use the **export** keyword to export types like functions, interfaces and classes

```
export class Person{
    constructor(public firstName: string,
                public lastName: string) { }
}
```


Importing from a Module

- Use the **import** keyword to import from another module

File: persons.ts

```
export class Person{
    constructor(public firstName: string,
                public lastName: string) { }
}
```

File: main.ts

```
import { Person } from './persons';

var p = new Person("Thomas","Huber");
```

Exporting multipe types: Using export on the types

File: persons.ts

```
export interface Names {
    firstName: string;
    lastName: string;
}

export class Person implements Names {
    constructor(public firstName: string, public lastName:
string) { }
}
```


Exporting multipe types: Using export statement

File: persons.ts

```
interface Names {
    firstName: string;
    lastName: string;
}

class Person implements Names {
    constructor(public firstName: string, public lastName:
string) { }
}
export { Person, Names }
```

Exporting multipe types: Using export statement

- You can even use aliases

File: persons.ts

```
...
export { Person as Friend, Names }
```

File: main.ts

```
import { Friend } from './persons';

var p = new Friend ("Thomas","Huber");
```


Import multiple types

- Just separate them by comma

```
import { Person, Names } from './persons';
```

- You can also give aliases here

```
import { Person as Friend, Names } from './persons';
```

- You can even import the full module

```
import * as Personas from './persons';
var p:Personas.Names = new Personas.Person("Thomas","Huber");
console.log(JSON.stringify(p));
```

Summary

- Modules are the way to go to structure your code
- Modules are based on file and folder structure
  – a file is a module, and it has exports and maybe imports
- Modules are native part of Node.js and ECMAScript2015
  – The browser's require a module loader like SystemJs

08_TypeScript_Decorators

What are Decorators?

- TypeScript's Decorators are like
  - Attributes in C#
  - Annotations in Java

```
@MyDecorator
class Person{ }
```

- Decorators allow you to annotate classes or members
- Currently the JavaScript-standard for Decorators is not finalized
  - that's the reason why they're an experimental feature in TypeScript
  - Angular relies heavily on decorators

trivadis
makes IT easier.

Turn the experimental Decorator feature on

- Either call the TypeScript-compiler with the option

```
tsc --target ES5 --experimentalDecorators
```

- Or add the option in your tsconfig.json-file

```
{
    "compilerOptions": {
        ...,
        "experimentalDecorators": true
    }
}
```


Decorators basics

- you can create and use a decorator on a
  - **class**
  - **property**
  - **method**
  - **accessor**
  - **parameter**
- A decorator is used in the form «**@expression**»
- expression must evaluate to a function with exact parameters
  - parameters depend where the decorator should be used (class, method etc.)

Property Decorators

- They have two parameters: the target-object and the propertyKey

```
function Format(target: any, propertyKey: string)
{
   console.log(propertyKey);
}

class Person
{
    @Format
    name:string;
}
```


Decorator functions

- To pass in data, create another function that returns the decorator function

```
function Format(myPrefix: string) {
    return function (target: any, propertyKey: string) {
        console.log(myPrefix + " my love");
    }
}

class Person
{
    @Format("Dear")
    name: string = "Thomas";
}
```

Class decorators

- Have just a single parameter: the constructor function

```
function EditorData(target:Function)
{

}

@EditorData
class Person{

}
```


Class constructor and decorator factories

```
function EditorData(data:EditorDataItem)
{
  return function TheReal(target:Function)
  {

  }
}

@EditorData({
    firstname:"Thomas",
    lastname:"Huber"
})
class Person{ }
```

But where do you store the data?

```
function EditorData(data:EditorDataItem)
{
  return function TheReal(target:Function)
  {
    // Where to store the EditorDataItem?
  }
}
```

- You could store it in a global array
  - but there's a better option: Reflect-metadata

Reflect-metadata: Store Metadata

- Reflect-metadata is a proposal for ES7. Package is in npm
  - https://www.npmjs.com/package/reflect-metadata
- Use it to store metadata

```
const EditorDataMetadataKey = "EditorDataKey";

function EditorData(data: EditorDataItem) {
    return function TheReal(target: Function) {
        Reflect.defineMetadata(EditorDataMetadataKey,
                               data,
                               target);
    }
}
```

Reflect-metadata: Read Metadata

```
@EditorData({
    firstname:"Thomas",
    lastname:"Huber"
})
class Person { }

let metadata = Reflect.getMetadata(EditorDataMetadataKey, Person)
                as EditorDataItem;

console.log(metadata.firstname);
console.log(metadata.lastname);
```


Angular uses Decorators intensively

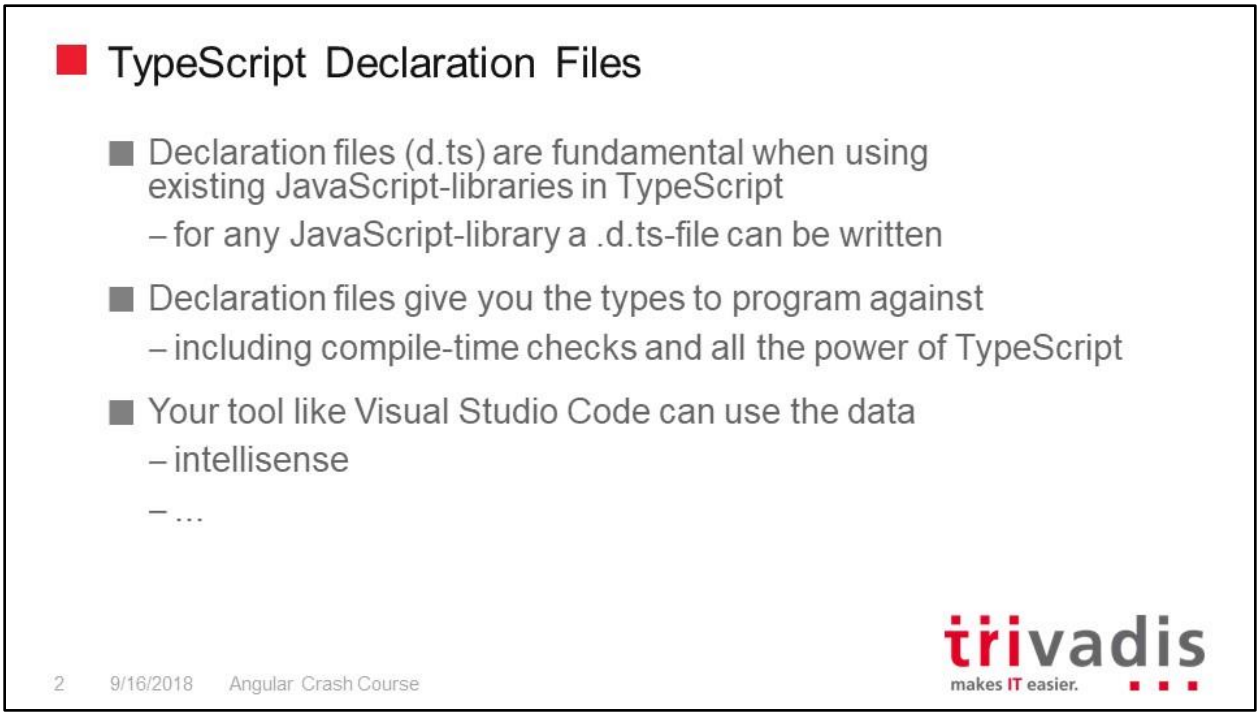
- Here's a Angular component
  - Note the @Component class decorator

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

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

Summary

- Decorators allow you to add metadata to
  – classes, methods, accessors, properties & parameters
- Decorators are still in experimental state
- Decorators are used intensively by Angular

09_TypeScript_DeclarationFiles



Declaration Files

- Search the available Declaration Files on http://microsoft.github.io/TypeSearch/
- Install the files via npm by using @types/library-name
- for example for the types for lodash use this command

```
npm install --save-dev @types/lodash
```

  - adds an entry to package.json (--save-dev-parameter)
  - adds the .d.ts-file to node_modules/@types/lodash

A sample for declarations

File: myJsLibrary.js

```
function printName(person)
{
  console.log(person.firstname);
  console.log(person.lastname);
}
```

File: index.html

```
<head>
        <title>TypeScript + Angular 2 Training</title>
        <script src="myJsLibrary.js"></script>
</head>
```

A sample for declarations

- Now let's assume you want to use the printName-method in your TypeScript-code

File: main.ts

```
let p = {firstname:"Thomas",lastname:"Huber"};

printName(p); // Works, but underlined in red
```

- The call works, but no types known by TypeScript
  - no intellisense
  - no compile-time checking
  - …

Let's declare the function

- Declaring the function and an interface, and TypeScript is happy! ☺
  - note the **declare** keyword and the function without implementation

File: main.ts

```
interface Person{
    firstname:string;
    lastname:string;
}
declare function printName(person:Person):void;

let p = {firstname:"Thomas",lastname:"Huber"};

printName(p);
```

Summary

- Declaration files (.d.ts) contain declarations for existing JavaScript libraries
- Since TypeScript 2.0 they're easily installable via npm

10_Angular_Introduction

Angular 2+

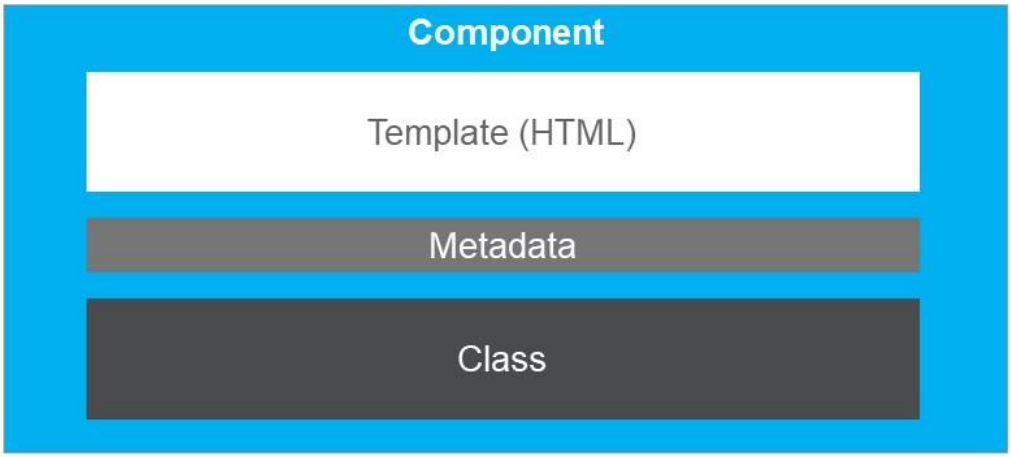
- Single-Page-Application (SPA) Framework by Google
  - is different to Angular 1.x: No controllers, no $scope
  - Angular is more than a Framework, it's a platform
- Completely rebuilt with TypeScript
- Use your HTML, CSS and TypeScript skills to build apps
  - Instead of TypeScript you could also use plain JavaScript or Dart

trivadis
makes IT easier.

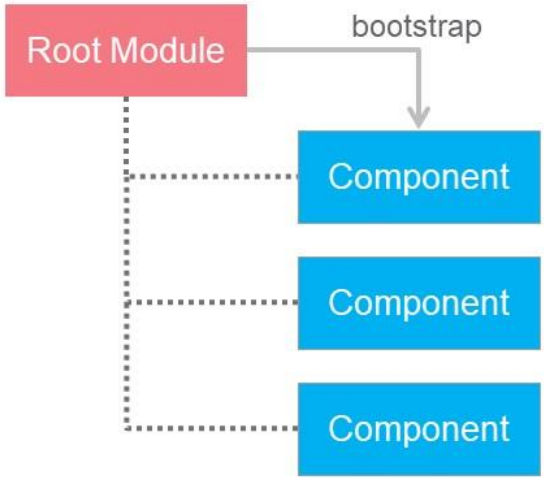
Why should you choose Angular?

- It's fast!
- It's clean!
- It has a powerful data-binding engine
- It is component-based and allows great structuring of apps

Angular Apps are component-based

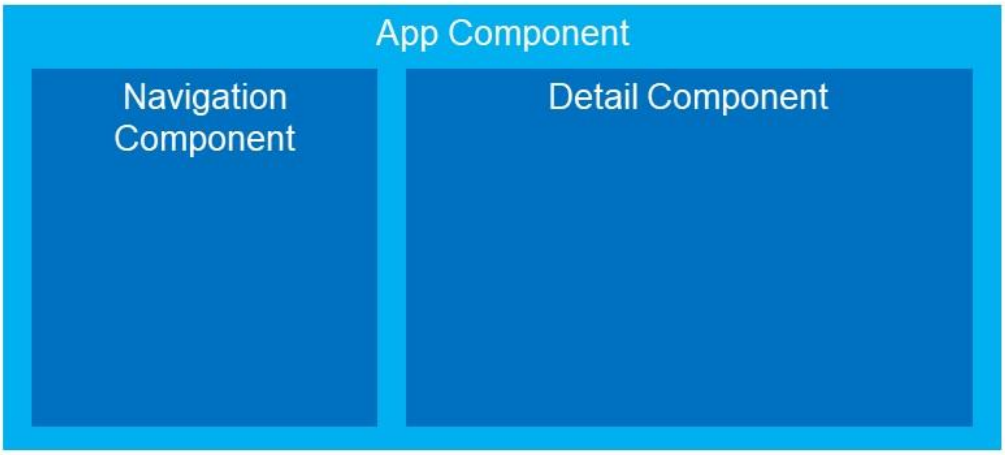


Components are structured into Angular-Modules



- Every Angular App has at least one module, the Root Module
- The module can bundle multiple components
- One Component must be bootstrapped for startup
  – only true for the root module

Components can be nested



Beside Components an app uses Services

- A service is a class that gets injected into components
  – for example a class that encapsulates access to a REST-Api

trivadis makes IT easier.

Summary

- Angular is component based
- Angular has its own Module-system
- Angular is built with TypeScript

trivadis makes IT easier.

11_Angular_SettingUpAnApplication

Setting up your environment

- Install node.js & npm
  - https://nodejs.org
- Install Angular CLI

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

- Install a development environment
  - Visual Studio Code: https://code.visualstudio.com/
  - Web Storm: https://www.jetbrains.com/webstorm/
  - ...

trivadis
makes IT easier.

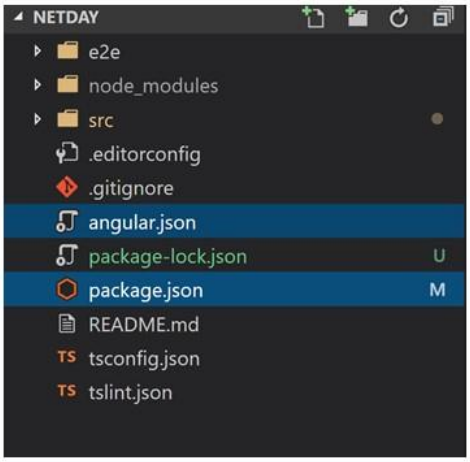
Angular CLI

- Command line interface for Angular
- Build an Angular application
- Generate files (Scaffolding)
- Execute / run the application
- Run unit and e2e tests
- Prepare and «optimize» the application for deployment

CLI: Generate new app

`ng new my-app`	Generate a new app in /my-app
`ng new my-app --dry-run`	Report without writing files
`ng new my-app --skip-install`	Generate without npm install
`ng new --style scss`	Use SCSS instead of CSS
`ng new my-app --routing`	Generate app with routing

Project structure



trivadis makes IT easier.

CLI: Generate

`ng generate <blueprint>`	Generate by schematics
`ng g c product-list`	Generate a component
`ng g c <n> --inline-template`	Template in ts file?
`ng g c <name> --spec=false`	Without unit tests
`ng g c <name> --flat`	Should a folder be created

trivadis makes IT easier.

CLI: Serve and build

`ng serve`	Serve the app
`ng s -o`	Serve the app and open
`ng build`	Build in dev mode
`ng build --prod`	Build in prod mode

CLI: Other commands

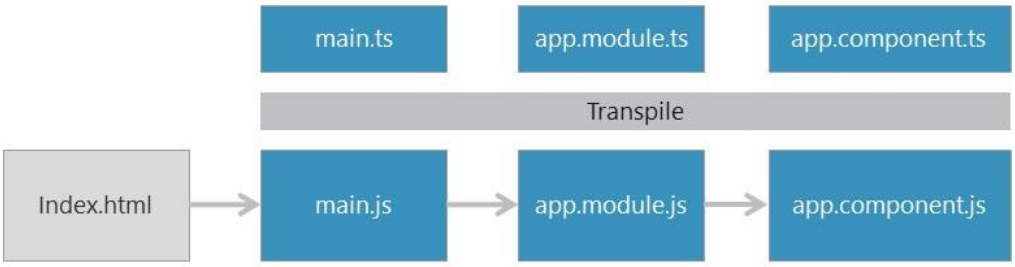
`ng test`	Run unit tests
`ng test --code-coverage`	Create code coverage doc
`ng e2e`	Run end-to-end test
`ng doc <filter>`	Open angular.io API reference

CLI: Other commands

`ng g application <name>`	Create a project in same ws
`ng g library`	Create a library
`ng add <package>`	Add package based on schematics
`ng update`	Update packages

Or create own schematics

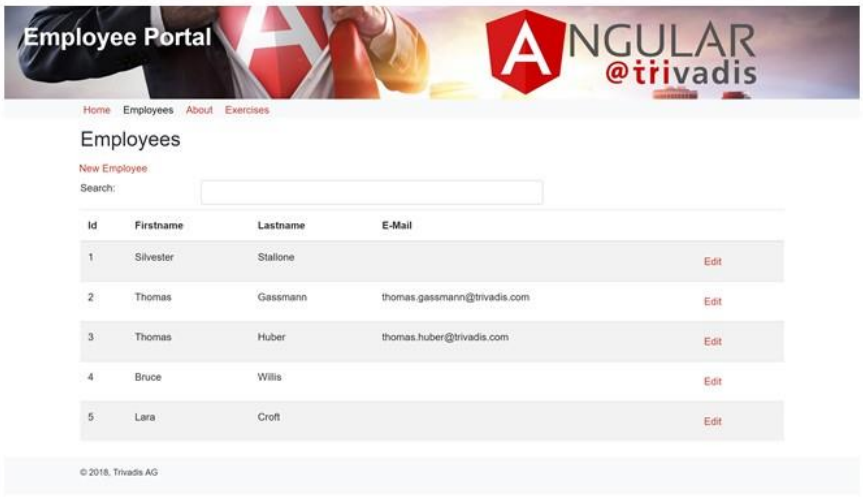
Architecture of the initial app



Try it:
Startup your initial angular project

trivadis makes IT easier.

- The app you build in this course: Employee Portal

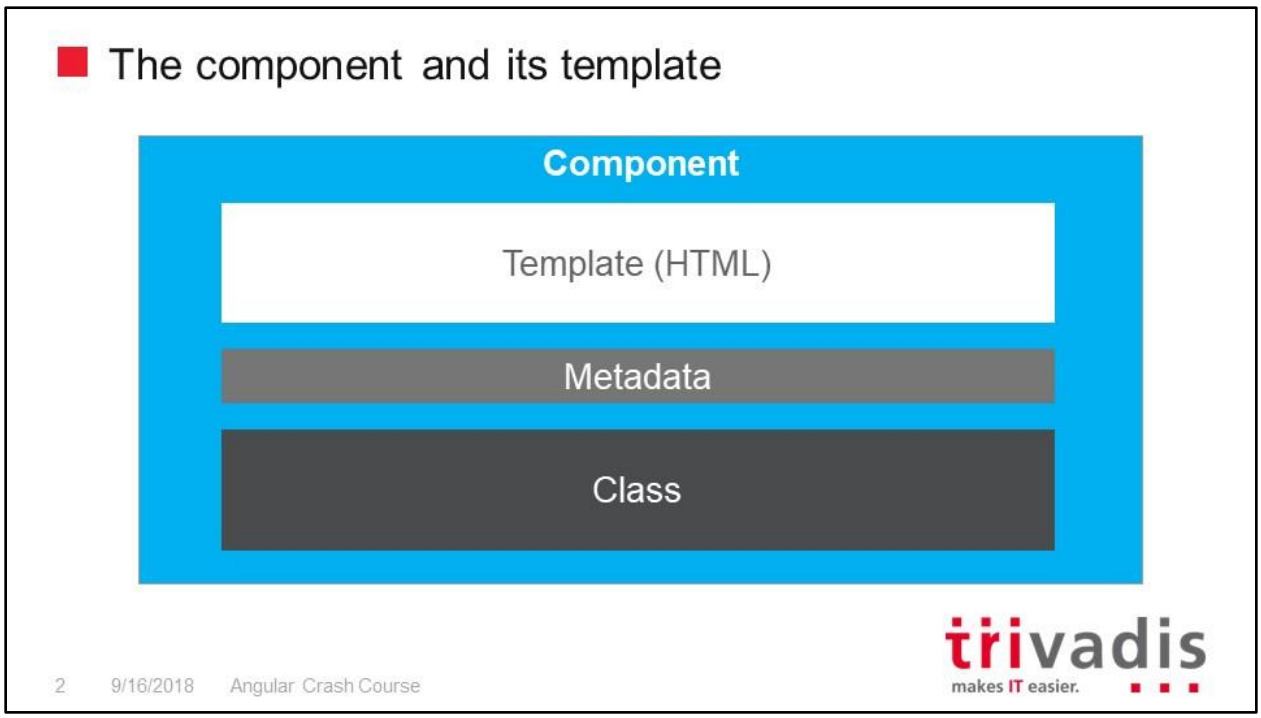


trivadis makes IT easier.

Summary

- An angular app has a root module
- The root module is usually bootstrapped in a main.ts-file
- To create a new project, the Angular CLI is the way to go

12_Angular_DataBinding



The component and its template

A component is a class decorated with the Component-decorator

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

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


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";
  }
}
```


Reference Elements in the Template

- Adding #name to an element in your template allows you to reference it

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

Reference Elements in the Template

- You can even access the element directly in the expression

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


TwoWay-Data Bindings: First trials

- You can use property-binding and event-binding in combination

```
<input #myInput type="text"
  [value]="fullname"
  (keyup)="onUpdate(myInput.value)">`
```

- Instead of accessing the value from the element, you can grab it from the event

```
<input type="text"
  [value]="fullname"
  (keyup)="onUpdate($event.target.value)">`
```

TwoWay-Data Bindings: Get rid of the onUpdate-method

```
 <input type="text"
   [value]="fullname"
   (keyup)="onUpdate($event.target.value)">`
```

- Instead of using the onUpdate-method, you can use an expression in the template to set the fullname-property

```
@Component({ ...,
  template: `... <input #myInput type="text"
            [value]="fullname"
            (keyup)="fullname = $event.target.value">`
})
export class AppComponent {
  fullname: string = "Lara Croft";
}
```


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: ngModel

```
<input type="text"
  [value]="fullname"
  (keyup)="fullname = $event.target.value">`
```

- Instead of accessing and remembering the value-property, you can use the ngModel-directive

```
<input type="text"
  [ngModel]="fullname"
  (keyup)="fullname = $event.target.value">`
```


TwoWay-Data Bindings: ngModelChange

```
<input type="text"
  [ngModel]="fullname"
  (keyup)="fullname = $event.target.value">`
```

- Instead of accessing the keyup event, you can use the *ngModelChange*-event.
  - The *$event*-variable contains directly the value (no «*.target.value*» is needed to access that value)

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

TwoWay-Data Bindings: the final syntax

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

- You can see above that there's 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»

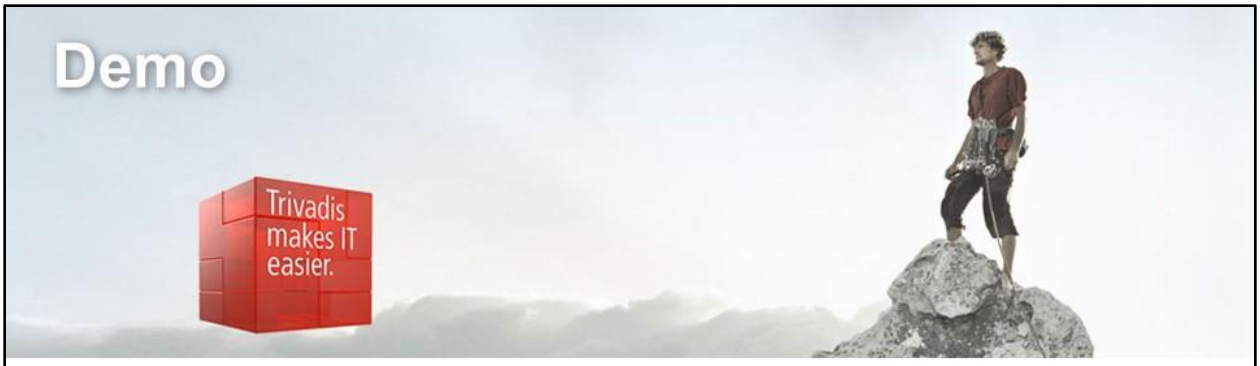
```
<input type="text"
  [(ngModel)]="fullname">
```


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 }
```

Demo



Show title in exercise/databinding/databinding.component.ts
Add Two Way Data Binding

trivadis
makes IT easier.

More Data Bindings

- Binding the presence of an «active»-class

```
<input [class.active]="isDeveloper == true"
```

- Binding a style-property

```
<input [style.width.px]="mysize"
```

- Binding to an attribute

```
<td [attr.colspan]="myColSpan"
```

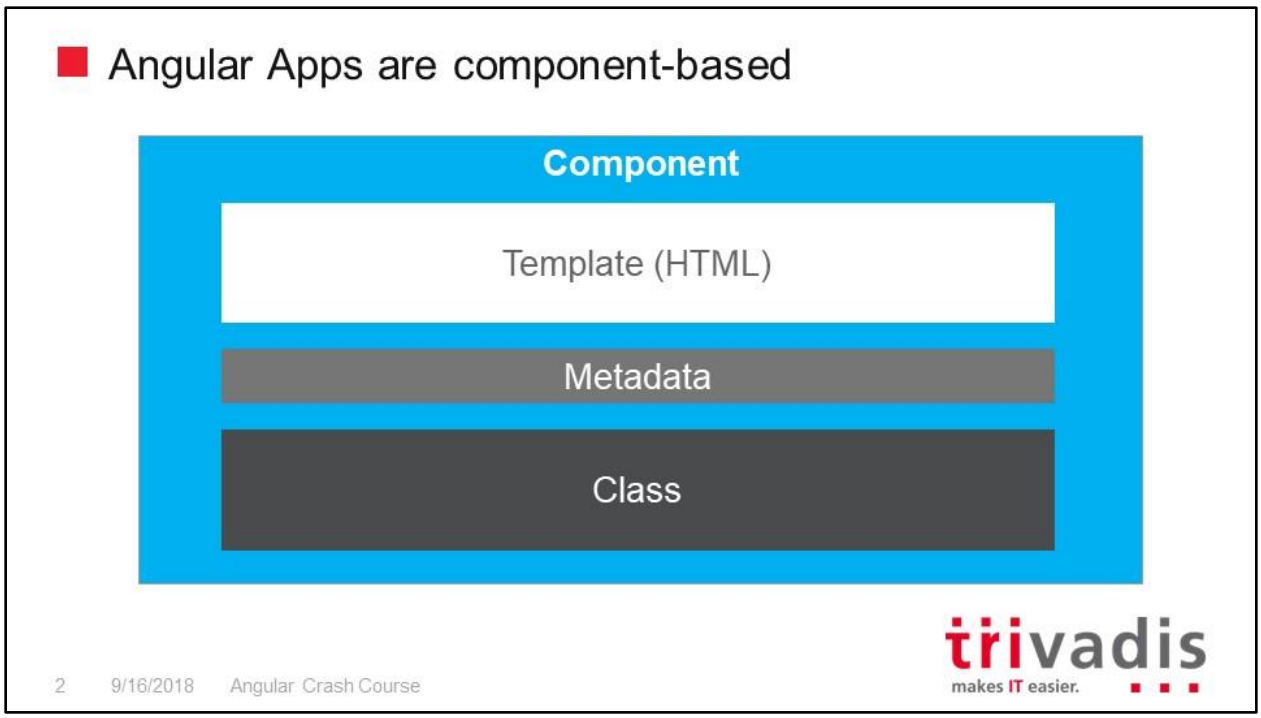
- Find more in the Angular cheat sheat on
  - https://angular.io/docs/ts/latest/guide/cheatsheet.html

trivadis
makes IT easier.

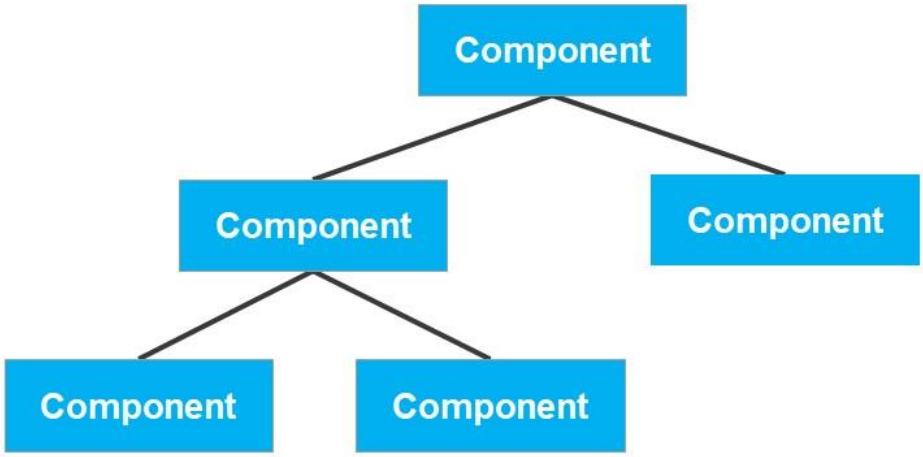
Summary

- Data Binding in Angular is powerful
  - {{ }} for displaying expressions
  - [] for binding properties
  - () for binding events
  - [()] for two-way bindings

13_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>
```

- 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
  - that means you can bind 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>
```


Defining Component events

- Use Angular's EventEmitter

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

@Component( ...)
export class PersonDetailComponent {
    ...
    remove = new EventEmitter<Person>();

    onRemove() {
       this.remove.emit(this.person);
    }
}
```

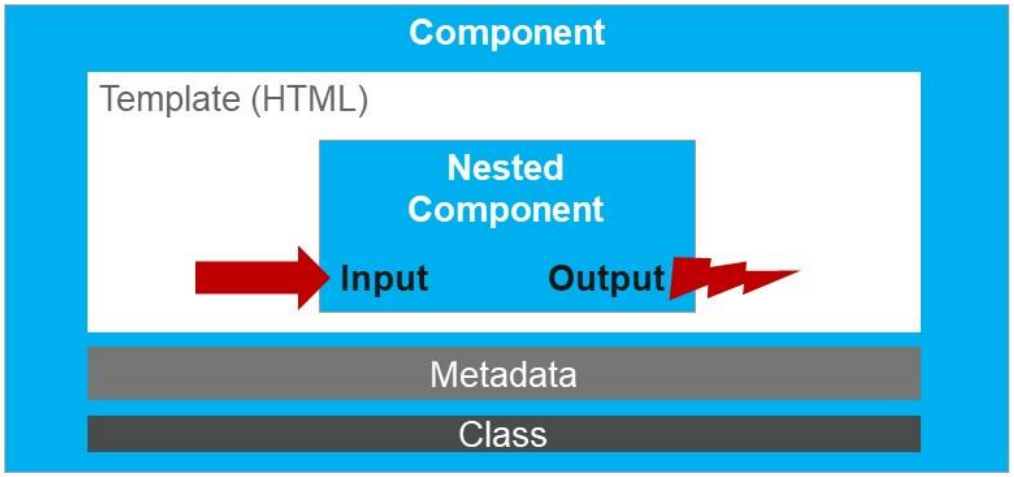
The Output of a Component

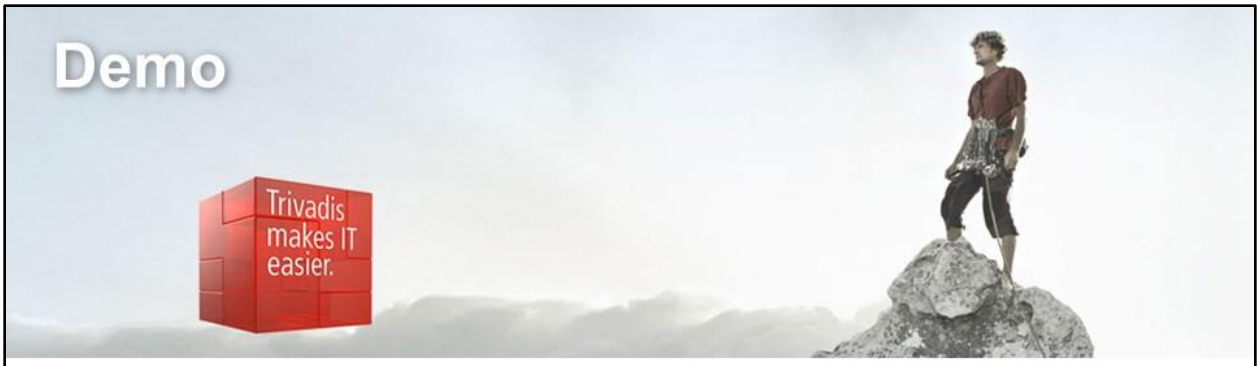
- To use the event from the outside, define it as Output!

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

@Component( ... )
export class PersonDetailComponent {
    ...
    @Output()
    remove = new EventEmitter<Person>();
}
```

```
<person-detail [person]="selectedPerson"
  (remove)="onRemove($event)"></person-detail>
```


Wrapup: Input and Output





Exercise/component/component.ts
Move selected person code to a detail-component (with Input)

Add a remove button to the new component and emit event out of the component (Output)

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']
})
```


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'); }
}
```

Summary

- Components need to be declared in an app module
- Components have their own HTML-selector
- Define the «API» of your component with Input / Output

14_Angular_Directives

What is a Directive?

- A Directive is an Angular Extension for classical HTML
- This can be a custom HTML element or attribute

trivadis
makes IT easier.

There are three directives in angular

- Components
  – are referenced by their selector (custom HTML tag)
- Structural Directives
  – are changing the DOM layout by adding/removing DOM elements
- Attribute Directives
  – change the appearance or behavior of an element

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

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;
}
```


ngIf

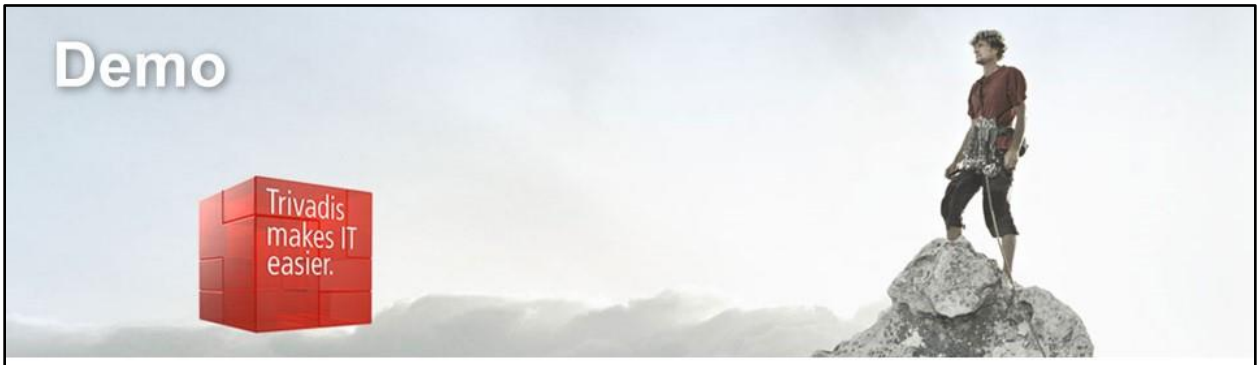
- div is only displayed if selectedPerson is not undefined/null

```
@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; ...
}
```

=> Exercise/ngfor/ngfor.component.ts
Create a table and fill it with the employees and "ngFor"-directive

=> Exercise/ngif/ngif.component.ts
Display the div with the firstname-input only if an employee is selected.
Use "ngIf" to do this!

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>
```


Attribute Directives

- **ngModel** – used for TwoWay data binding
- **ngStyle** – to bind to multiple styles
- **ngClass** – to active multiple classes
- …

ngStyle

- [style.property] is just for a single property, ngStyle is for multiple

```
@Component({...
  template: `<input type="text" [ngStyle]="getMyStyles()" ...>`
})
export class AppComponent {
  getMyStyles() {
    let styles= {
                  "background-color":"black",
                  "color":"white"
                };
    return styles;
  }
}
```


ngClass

- [class.className] is just for a single class, ngClass is for multiple

```
@Component({...
  template: `<input type="text" [ngClass]="getMyClasses()" ...>`
})
export class AppComponent {
  getMyClasses() {
    let classes = {
                    "isActive":"true",
                    "isSelected":"true"
                  };
    return classes;
  }
}
```

How to create an attribute directive

- The most important property of an attribute directive is that it only modifies its own host element

```
import { Directive } from '@angular/core';
 …

@Directive({ selector:"[appSelect]" })
export class CardHoverDirective {

  constructor(private el: ElementRef) {
     el.nativeElement.style.backgroundColor = "gray";
  }
}
```


Register it in ngModule

- Register all directives separately

```
import { SelectDirective } from './directives/select.directive';

@NgModule({
  imports: [CommonModule],
  exports: [CommonModule,
    SelectDirective],
  declarations: [SelectDirective],
  providers: []
})
export class SharedModule {}
```

Selectors

- Selectors is css attribute that identifies a component in a template
- [] make an attribute selector. Angular locates each element in the template that has an attribute name 'appSelect' and applies the logic of this directive to that element

```
@Directive({ selector: '[appSelect]' })
export class SelectDirective {}
```


Types of selectors

- element-name: select by element name
- .class: select by class name
- [attribute]: select by attribute name
- [attribute=value]: select by attribute name and value
- :not(sub_selector): select only if the element does not match the subselector

Directives Input

- The @Input decorator gives you the option to set some property value while applying the directive.

```
@Directive({ selector: '[appSelect]' })
export class SelectDirective {
  @Input() backgroundColor: string;
   constructor(private el: ElementRef, private renderer:
   Renderer){}
```

- Use it in a component

```
<tr *ngFor="let employee of employees$ | async"
          appSelect [backgroundColor]="'gray'">
```


@Hostbinding and @HostListener decorators

- @HostBinding
  - lets you set properties on the element or component that hosts the directive
- @HostListener
  - lets you listen for events on the host element or component.

@HostBinding: possible parameters

- propertyName: references a property of the host with a given property name
- attr.[attributeName]: references an attribute of the host with a given attribute name. Using the null value removes the attribute from the HTML element.
- style.[styleName]: references a property to a style of the HTML element
- class.[className] references a property to a class name of the HTML element.

@HostBinding

```
@Directive({ selector: '[appSelect]' })
export class SelectDirective {

  @HostBinding('class.selected') isSelected = false;

   constructor(private el: ElementRef, private renderer:
        Renderer) {}

   @HostListener('mouseover')
   @HostListener('mouseleave')
   setSelected() {
       this.isSelected = !this.isSelected;
    }
}
```

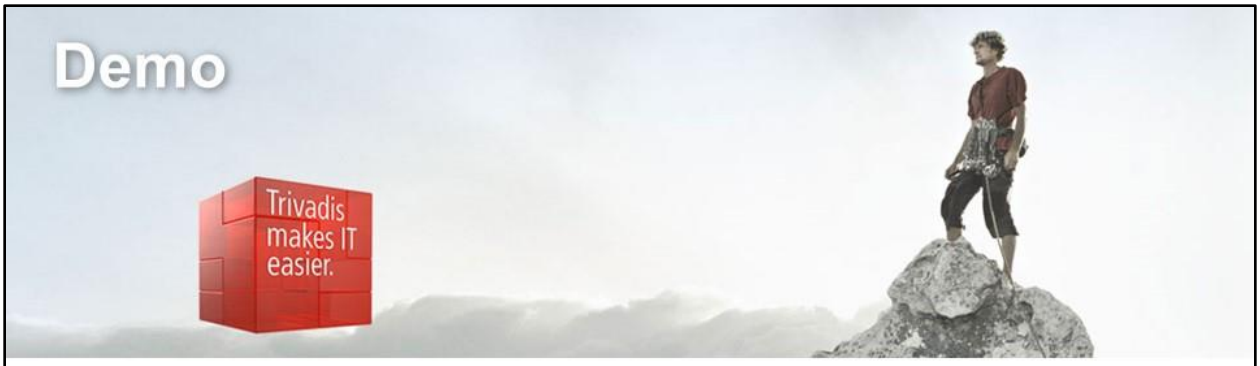
@HostListener

```
@Directive({ selector: '[appSelect]' })
export class SelectDirective {

 constructor(private el: ElementRef, private renderer:
       Renderer) {}
 @Input() backgroundColor: string;

 @HostListener('dblclick')
  doubleClick() {
  this.renderer.setElementStyle(this.el.nativeElement,
       'backgroundColor', this.backgroundColor);
 }
}
```


Summary

- Directives extend HTML with the power of Angular
- there are 3 kinds of directives (Components, Structural, Attribute)
- If you want, you can even write your own directive by using the @Directive-decorator
- Use @Hostbinding to bind to the input properties of a host element from within a directive
- User @Hostlistener to listen to the output events

Demo



Create a new Attribute Directives (e.g. SelectDirective, create it under shared/directives/)

Use it inside the employee-list.component.ts

Add a **`HostBinding`** and **`HostListener`** functionality

trivadis
makes IT easier.

15_Pipes

What is a pipe in Angular?

- A pipe is used to transform or filter data that is displayed in HTML
- Useful when doing transformations repeatedly
- There are Built-in pipes (UpperCasePipe, CurencyPipe, DatePipe, PercentPipe, DecimalPipe, JSONPipe, AsyncPipe)

trivadis
makes IT easier.

Date pipe explained

```
export class NumberPipeComponent {
        currentDate : Date = new Date();
}
```

```
<p> Today is {{ currentDate| date:"MM/dd/yy" }} </p>
```


Decimal pipe explained

- number_expression | number[:digitInfo[:locale]]
- format: {minIntegerDigists}.{minFractionDigists}-{maxFractionDigits}

```
@Component({
 selector: 'number-pipe',
 template: `<div>
              <!--output '012.63847'-->
              <p>num1 3.2-5: {{num1 | number: '3.2.-5'}}</p>
           </div>`
}
export class NumberPipeComponent {
       num1: number = 12.638467846
}
```

How to create a custom pipe

```
@Pipe({
 name: 'employeeFilter'
})
export class EmployeeFilterPipe implements PipeTransform {

transform(value: Employee[], filterBy: string): Employee[] {
        filterBy = filterBy ? filterBy.toLocaleLowerCase() :
        null;
        return filterBy ? value.filter((e: Employee) =>
        (e.firstname.toLocaleLowerCase() +
        e.lastname.toLocaleLowerCase()).indexOf(filterBy) !== -1)
        : value;
 }
}
```


How to create a custom pipe

```
<input type="search" [(ngModel)]="listFilter" class="form-
control" />

<tr *ngFor="let employee of employees$ | async |
        employeeFilter:listFilter" appSelect>
<td>{{ employee.id }}</td>
<td>{{ employee.firstname }}</td>
<td>{{ employee.lastname }}</td>
<td>{{ employee.email }}</td>

</tr>
```

Pure pipes

- Pure pipes are executed by Angular when a pure change to the input was detected. This can be a primitive input value such as String, Number, Boolean, Symbol or object reference (Array, Function, Object)
- However, changes within objects are ignored. This means changing an object property, modifying an array or chaning input data will not call a pure pipe logic.

Inpure pipes

- Inpure pipes, on the another hand, are raised during every component change detection cycle. They are called more often than pure pipes. For example by every key down or mousedown event.

```
@Pipe({
  name: 'employeefilter',
  pure: false
})
export class EmployeeFilterPipe extends PipeTransform {}
```

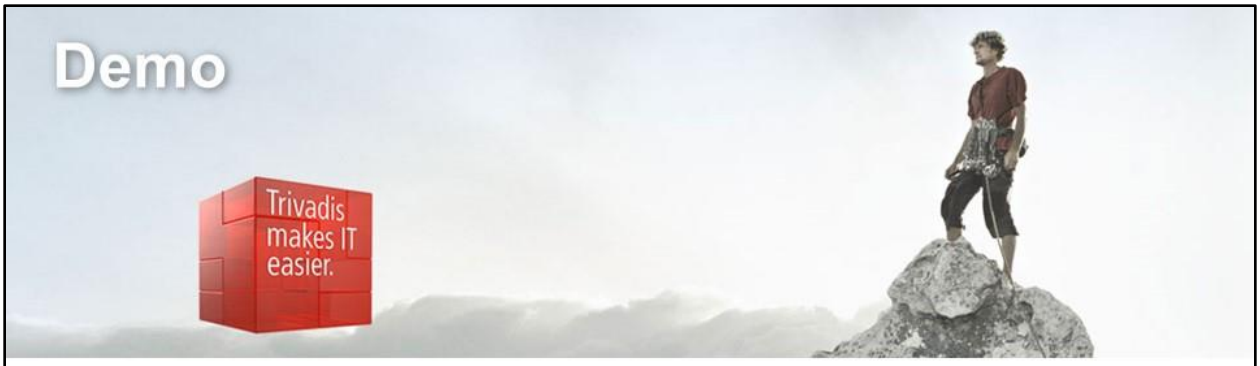
Problems with custom FilterPipe and OrderByPipe

- There are no built in filtering or sorting pipes in Angular on purpose (in contrary to AngularJS). The reason is their poor performance.
- Angular calls impure pipes in every change-detection cycle
- Avoid using them as filtering and sorting are expensive operations

Summary

- Pipes enable you to transform data to be displayed in templates
- Angular comes with many built-in pipes. (Currency, JSON, Decimal, Percent, Date and more…)
- Pure pipes perform much better than inpure pipes a you should consider using inpure pipes

Demo



Create a pipe (e.g. FilterEmployeePipe)

Use it inside a component

trivadis
makes IT easier.

16_Modules

What is an Angular module?

- The main purpose of a module is to combine and organize code with similar logical context into one unit
- We can extend our application logic by importing external modules (libraries)
- Modules make it possible to load some funcionality on demand (lazy loading)
- There are 2 types of modules, root and feature modules
- A module can be loaded eagerly when the application starts or lazyly (assynchronously) by the router

trivadis
makes IT easier.

Root Module

- Every application has at least one module – the root module. It is used to bootstrap your application
- You usually name it AppModule

Feature Modules

- A feature modules purpose is to organize your code and achieve modularity
- Feature modules should only import services from RootModule.
- You should lazy load feature modules whenever possible. Especially for performance reason

Declaration of a module

- The @NgModule decorator defines an Angular Module
- Module configuration:
  - imports
  - exports
  - declarations
  - entryComponents
  - providers
  - bootstrap

Module Bootstraping

- **Imports** array: contains a list of another modules that needs to be imported for your module
- **Exports** array: contains a list of reausable components, directives, pipes in your module that you want to user outside of your module
- **Providers** array: contains a list of services that your app needs
- **Declarations** array: tells Angular which components, directives and pipes (declarables) belong to that module. A declarable can only belong to one module. It is used inside a template
- **Bootstrap** array: Root components that are launched on bootstrapping

Declaration of a module

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

@NgModule({
   imports: [AboutRoutingModule],
   exports: [],
   declarations: [AboutListComponent],
   providers: []
})
export class AboutModule {}
```


Lazy loading modules

- By lazy loading modules Angular creates a child injector for specified providers
- Registering a specified service within a child module can lead to unintended consequences, as more instances of the same service would be created
- This might not be wrong, but sometimes you may need to assure that you get only one instance of a given service e.g. authentication service which delivers the same information across the whole application

Lazy loading modules

- To ensure that a module will be loaded on demand, we replace the component property with the loadChildren property in the route definition and we pass a string instead of a symbol as a path to the module. We define the class of the module as well.

```
const routes: Routes = [
 { path: '', redirectTo: 'eager', pathMatch: 'full' },
 { path: 'eager', component: EagerComponent },
 { path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }
];
```


Dealing with instances

- Loading services multiple times can be avoided by defining a custom static forRoot() or forChild() method that returns a configuration object that defines the services we want to export

```
export class CoreModule {
static forRoot(): ModuleWithProviders {
        return {
        ngModule: CoreModule,
        providers: [MyService]
  };
 }
}
```

Dealing with instances

- To register a feature module in the root application module we call the forRoot() method defined for example in CoreModule.

```
@NgModule({
   imports: [
        CoreModule.forRoot()
   ]
})
```


Dealing with instances

- forRoot() should only be called once in the AppModule. To make sure it is not called multiple times, you can check it inside the constructor

```
export class CoreModule {
  constructor(@Optional() @SkipSelf() parentModule: CoreModule)
{
    if (parentModule) {
      throw new Error(`Core has already been loaded. Import Core
                          modules in the AppModule only.`);
    }
  }
}
```

Shared modules

- Shared modules are meant to share functionality across your application
- All the „dumb“ components and pipes should be implemented as a shared modules
- A shared module should not import and inject services from the root module

How to create a module with Angular CLI

- ng generate module [name].
  Creates a new NgModule with given name.
- ng g module my-module --routing.
  With the routing option, a separate file my-module-routing.module.ts will be created, where module routes can be specified.

Outlook Angular V7

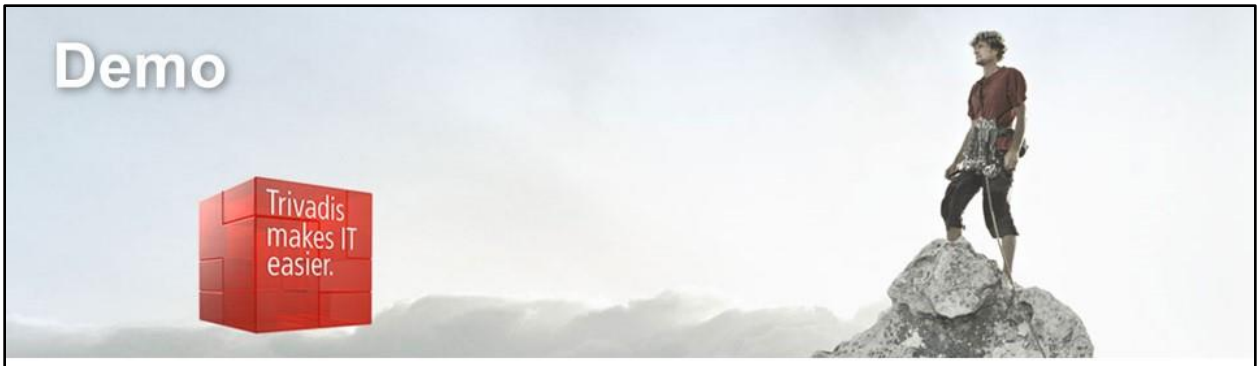
Statement from Igor Minar (Angular Team Lead)

"But we made progress across the board, which unlocked new possibilities, and we now believe that we'll be able to keep the guarantees and features without NgModules in the (near) future. I'm *hoping* in v7."

Summary

- Modules allows you to organize and streamline your code.
- You can put commonly used directives, pipes and components into one module and import just that module
- Modules can be loaded eagerly or on lazyly.
- By calling a custom forRoot() method in the application module you can make sure that services inside this module will be created as singletons and can be used accross other modules

Demo



Create a Employee Module

Place all employee components inside this module

Make sure Core Module is only loaded inside AppModule and services are singleton

trivadis
makes IT easier.

17_Angular_ServicesAndDependencyInjection

What is a Service

- A service is just a class that has a specific purpose
  - loading data
  - logging
  - …

```
export class PersonDataService {
    loadPersons(): Person[]
    {
       return  [{firstname: 'Silvester', lastname: 'Stallone'},
                {firstname: 'Bruce', lastname: 'Willis'}];
    }
}
```

What is Dependency Injection?

- Classes take instances of other classes (the dependencies) as constructor parameter
  - The instances are «injected» into the class
- Dependency Injection leads to loose coupling
  - and to testable code
- Angular has a built in Dependency Injector!

Inject a Service

- **Step 1:** Mark the Service as Injectable

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

@Injectable()
export class PersonDataService {
    loadPersons(): Person[]
    {
       return  [{firstname: 'Silvester', lastname: 'Stallone'},
                {firstname: 'Bruce', lastname: 'Willis'}];
    }
}
```

Inject a Service

- **Step 2:** Register the Service as a provider
  - either in the providers property of the module or of a component
  - when registered on a component, it will be available on all nested child-components

```
import { PersonDataService } from './person-data.service';

@Component({
  selector: 'my-app',
  template: `...`,
  providers: [PersonDataService]
})
export class AppComponent { }
```


Inject a Service

- **Step 3:** Use the service
  - Just add a constructor parameter, Angular will inject it for you

```
import { PersonDataService } from './person-data.service';

@Component(...)
export class PersonListComponent {
    persons: Person[];
    constructor(private _personDataService: PersonDataService) {
       this.persons = _personDataService.loadPersons();
    }
}
```

Use lifecylce hook to load data

- **Step 4 / optional:** Use OnInit to load the data

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

@Component(...)
export class PersonListComponent implements OnInit {
    persons: Person[];
    constructor(private _personDataService: PersonDataService){}

    ngOnInit() {
        this.persons = this._personDataService.loadPersons();
    }
}
```


Summary

- Angular has an integrated Injector for dependencies
- Mark your services with the @Injectable-decorator
- Create your providers on the module or on a component
  – use the component hierarchy to control the lifetime of your service

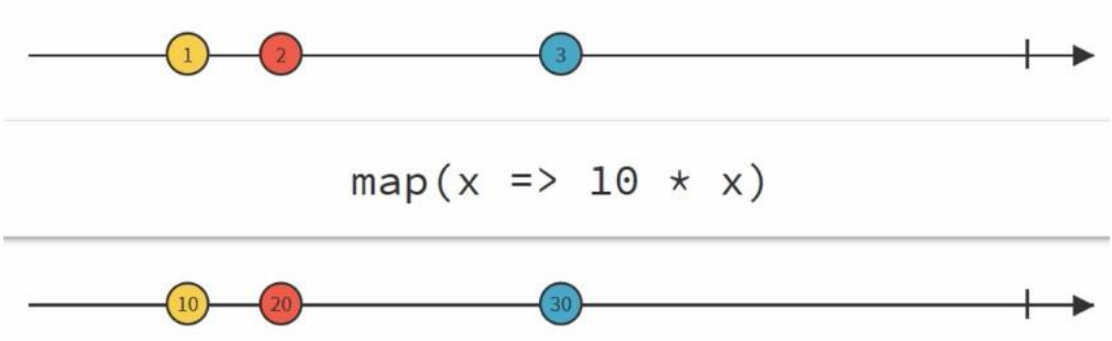
18_Http

HTTP in Angular

- Angular has it's own http module
- To read the data, you can either use
  - promises
  - observables (rxjs = Reactive Extensions)
- Observables are a powerful way to work with async data
  - Angular's http module uses observable by default

trivadis
makes IT easier.

Observables



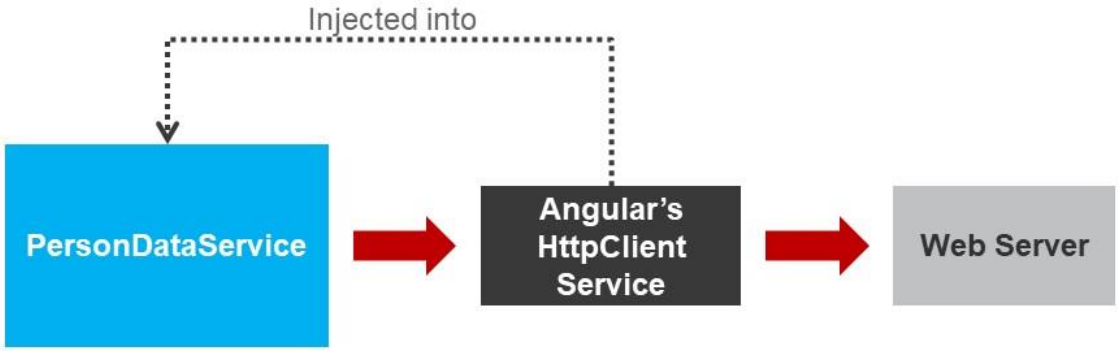
trivadis makes IT easier.

HttpClient

- In Angular 4.3, a new module for accessing REST Services has been introduced.
- It is a new implementation of the former *HttpModule*
- The new *HttpClient* service was introduced
- The old service Http is deprecated since Angular 5
- Working with Observables is the default. In Angular 5, the **pipe** operator was introduced

trivadis makes IT easier.

Using HttpClient



trivadis makes IT easier.

Using HTTPClient

- **Step 1:** Register the Http Service Provider => The HttpModule

```
...
import { HttpClientModule } from '@angular/common/http';
...
@NgModule({
  imports: [CommonModule],
  exports: [CommonModule,
        HttpClientModule],
  declarations: [...],
  providers: [...],
})
export class SharedModule { }
```


trivadis makes IT easier.

Using HTTPClient

- **Step 2:** Get the Http-Service via Dependency Injection

```
…
import { HttpClient } from '@angular/common/http';

@Injectable()
export class EmployeeService {
    constructor(private _httpClient: HttpClient) { }
…
}
```


Using HttpClient

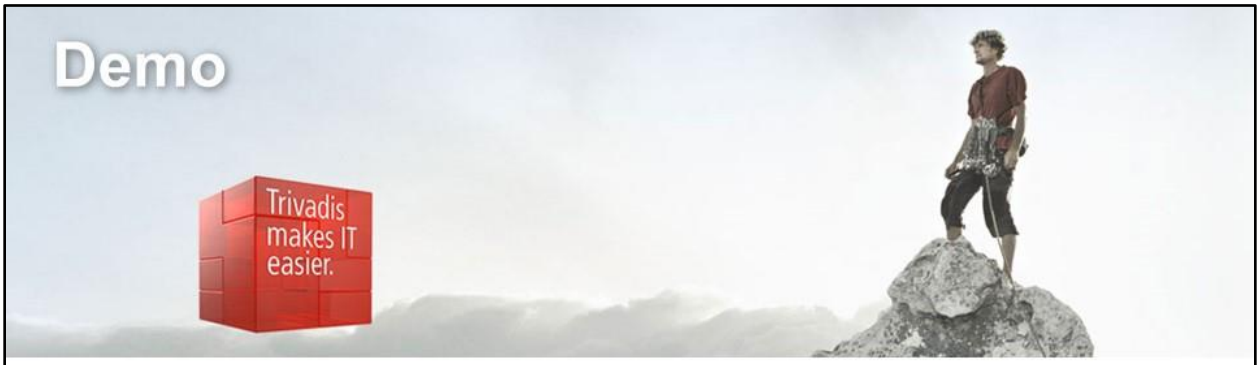
- **Step 3:** Send the request from the service

```
import { HttpClient } from '@angular/common/http';
@Injectable()
export class EmployeeService {
    constructor(private _httpClient: HttpClient) { }

     getEmployees(): Observable<Employee[]> {
        return this.http
          .get<Employee[]>(`${environment.apiBaseUrl}/employees`)
          .pipe(catchError((error: any) =>
                      Observable.throw(error)));
      }
}
```

Using HTTP with Observables

Step 4: Use the service

```
export class EmployeeListComponent implements OnInit {
    employees: Employee[];
    constructor(private employeeService: EmployeeService){ }

    ngOnInit() {
        this.employeeService.getEmployees()
        .subscribe(list => this.employees = list,
               error => alert(error)
        );
    }
}
```

Implement Method «getEmployees()» in Service EmployeeService.
Use URL: `` `${environment.apiBaseUrl}/employees` ``

See how the service is used inside EmployeeListComponent

Interceptors

- Interceptors are sitting in between your application and the backend.
- By using interceptors you can transform a request coming from the application before it is actually submitted to the backend.

Interceptors

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  intercept (req: HttpRequest<any>, next: HttpHandler):
       Observable<HttpEvent<any>> {

    const authReq = req.clone({
      headers: req.headers.set('Authorization', 'Bearer: XYZ')
    });
    return next.handle(authReq);
  }

}
```

Providing the Interceptor

```
@NgModule({
  imports: [CommonModule],
  exports: [CommonModule, HttpClientModule],
  declarations: [...],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptor,
      multi: true
    }
  ],
})
export class SharedModule { }
```


Global Error Handler

- Create a global http error handler
- http handle is an observable
- catchError was introduced with lettable operators

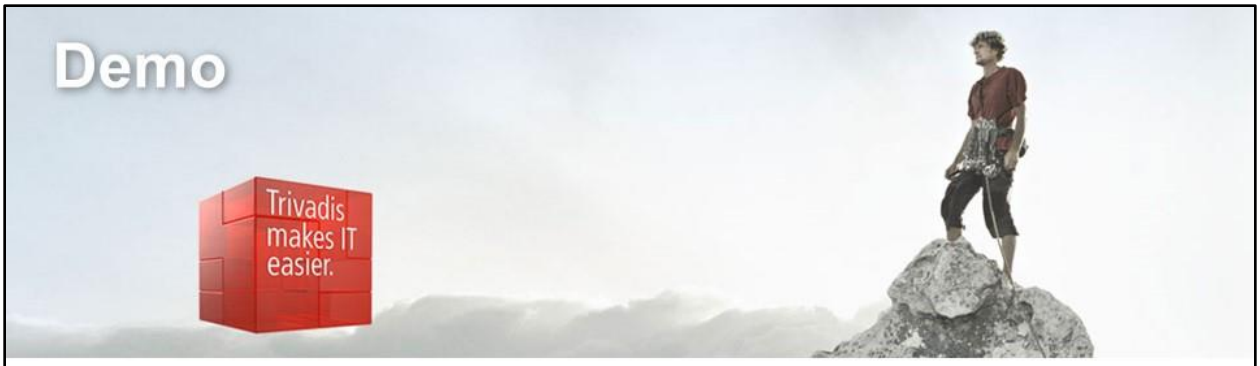
Global Error Handler

```
intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      catchError((error, caught) => {
        if (error instanceof HttpErrorResponse) {
          const httpError: HttpErrorResponse = error;
          console.log(`${httpError.message}`);
        }

        // Important! Always rethrow it
        return Observable.throw(error);
      })
    );
  }
```


Summary

- Angular's HttpClient-Services are in @angular/common/http
- You can use Promises or Observables
- Observables are especially powerful if you chain multiple calls

Demo



Create a new interceptor

Create a global http error handler

trivadis
makes IT easier.

19_Routing

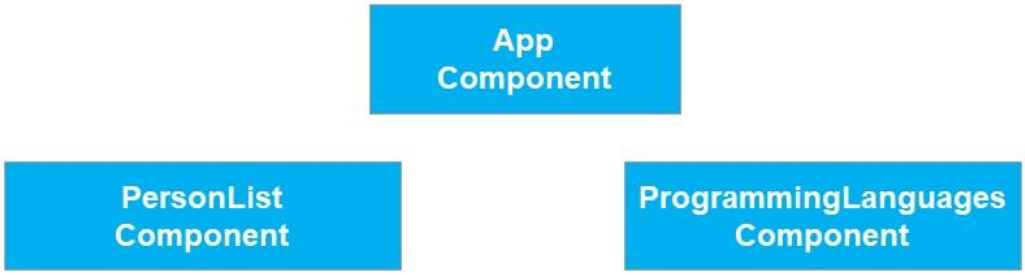
What is Routing?

- Angular is a Single Page Application (SPA) framework
- Somehow users need to navigate
- Navigation means Routing the user to different components

trivadis
makes IT easier.

Setting up Routing: The Component Structure

- Let's assume we have this component structure
- In the AppComponent, we want to have links to PersonListComponent and ProgrammingLanguagesComponent



trivadis makes IT easier.

Setting up Routing: Adding the base tag

- **Step 1:** Add the base tag to your index.html-file

```
<head>
  <base href="/">
  ...
</head
```

- Behind the scenes, Angular's router uses the browser's **history.pushState** for navigation
  - This allows you to make the URL-path look like you want

trivadis makes IT easier.

Setting up Routing: The Service Provider

- **Step 2:** Setup the Router service provider
  - call the RouterModule's forRoot-method to create a provider

```
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [BrowserModule, FormsModule, RouterModule.forRoot([
  { path: 'languages', component: ProgrammingLanguagesComponent},
  { path: 'persons', component: PersonListComponent }
    ])],
  ...
})
export class AppModule { }
```

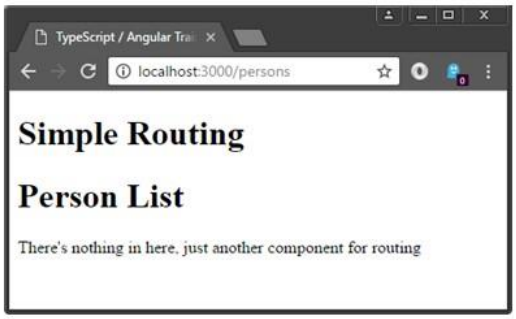

Setting up Routing: Add the router-outlet component

- **Step 3:** Add the router-outlet component
  - It comes with the router library @angular/router
- This is where the linked components will be displayed

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

@Component({
  selector: 'my-app',
  template: `<h1>Simple Routing</h1>
            <router-outlet></router-outlet>
            `
})
export class AppComponent { }
```

Test the routes

- Now the routes work already
  - **/languages** and **/persons**



Setting up Routing: Adding links

- **Step 4:** Adding router links

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

@Component({
  selector: 'my-app',
  template: `<h1>Simple Routing</h1>
            <a routerLink="/languages">languages</a>
            <a routerLink="/persons">persons</a>
            <router-outlet></router-outlet>
             `
})
export class AppComponent { }
```

Setting up Routing: Adding a redirect in the config

- **Step 5:** Redirect the root URL to **/persons**

```
@NgModule({
  imports: [BrowserModule, FormsModule, RouterModule.forRoot([

     { path: 'languages',
       component: ProgrammingLanguagesComponent},
     { path: 'persons', component: PersonListComponent },
     { path: '', redirectTo: '/persons', pathMatch: 'full'},
     { path: '**', redirectTo: '/persons', pathMatch: 'full'}

    ])],
```


Parameterized Routes

- If you navigate to a detail component, you might want to pass an id

Parameterized Routes: Setting up the route

- **Step 1:** Setting up the route with a parameter

```
const routes: Routes = [
  { path: 'employees', component: EmployeeListComponent },
  { path: 'employees/new', component: EmployeeComponent },
  { path: 'employees/:employeeId', component: EmployeeComponent }
];
```


Parameterized Routes: Navigate to the details

- **Step 2:** Navigate to the details

```
import { Router } from '@angular/router';
...
@Component(...)
export class EmployeeListComponent implements OnInit {
   ...
   constructor(private _service: EmployeeService,
      private _router: Router) { }

   onPersonClick(e: Employee) {
          let link = ['/employee', e.id];
          this._router.navigate(link);
    }
}
```

Parameterized Routes: Get parameter in details

Step 3: Grabbing the parameter

```
export class EmployeeComponent implements OnInit {
  employee$: Observable<Employee>;

  constructor(private route:ActivatedRoute,
    private service: EmployeeService){}

    ngOnInit() {
      this.route.paramMap.subscribe((p) => {
        const eId = +p.get('id');
        this.employee$ = this.service.getEmployee(eId);
      }
    }
}
```


Parameterized Routes: Navigate back from the detail

```
import { Location } from '@angular/common';

@Component(...)
export class PersonDetailComponent implements OnInit {
  person: Person;
  constructor(..., private location: Location){}
  ...
  goBack() {
    this.location.back();
  }
}
```

Refactor routing logic into a RoutingModule

- If you put all the routing into your AppModule, it might get confusing
- Move the routing code to a separate Module
- Then you just import that separate Module into the AppModule

Importing Angular Router in Feature Module

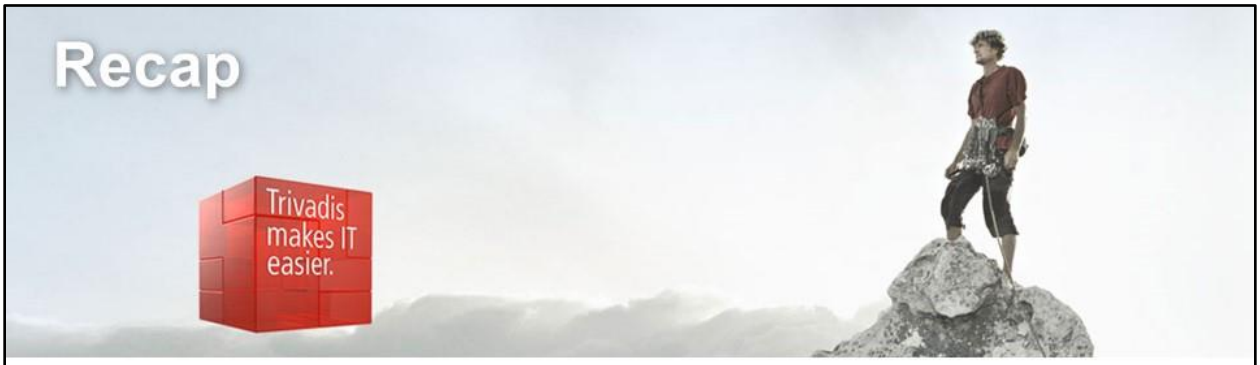
- The Router Service can only be imported once. (forRoot-Method)
- Routing inside a feature module is different.
- Import this module before calling forRoot()

```
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [SharedModule,
    RouterModule.forChild([…])
  ],
  ...
})
export class FeatureModule { }
```

Defining a Routing Module

- Better organization and easier to find
- Separation of concerns
- <NAME>-routing.module.ts

```
const routes: Routes = […];

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

Show how it is implemented in demo app

Query Parameters

- Can be important for search url
- We won't define a route for each search field

trivadis makes IT easier.

Query Parameters: Defining

- **Step 1: Define query params**

```
<a [routerLink]="['/employees',  employee.id]"
[queryParams]="{filterBy: listFilter}">Edit </a>
```

```
this.router.navigate(['/employees', id], {
      queryParams: { filterBy: this.listFilter }
});
```


trivadis makes IT easier.

Query Parameters: Retaining query params

Step 2: Add attribute when navigating back

```
<a [routerLink]="['/employees']"
[queryParamsHandling]="'preserve'">Back</a>
```

```
this.router.navigate(['/employees'], {
       queryParamsHandling: 'preserve'
});
```


Query Parameters: Reading

Step 3: Reading query parameters

```
constructor(
    private route: ActivatedRoute,
    private router: Router
  ) {}

  ngOnInit() {
    this.listFilter =
                this.route.snapshot.queryParams['filterBy'] || '';
  }
```

Providing Data with a Route

- Passing data to a route with the data property

```
const routes: Routes = [
  { path: 'employees', component:EmployeeListComponent },
  { path: 'employees/new', component:EmployeeComponent },
  {
    path: 'employees/:employeeId',
    component: EmployeeComponent,
    data: { title: 'Employee' }
  }
];
```

```
this.route.snapshot.queryParams['title'];
```


Providing Data with a Route Resolver

- Problem that a page loads and after that id loads the data. So it is just partially loaded
- This can be avoided by using a router resolver

```
@Injectable()
export class ProductResolver implements Resolve<IProduct> {
        constructor(private productService: ProductService,
                private router: Router) { }

        resolve(route: ActivatedRouteSnapshot,
                state: RouterStateSnapshot): Observable<IProduct>{
             let id = route.paramMap.get('id');
             // load data and return it
        }
}
```

Add a resolver to a Route

- The Resolver can be added to the route definition

```
const routes: Routes = [
…
{
        path: ':id',
        component: ProductDetailComponent,
        resolve: { product: ProductResolver }
}
];
```


Reading data from a resolver

- Instead of reading from a snapshot, reading from Observables is much better to get informed about route changes

```
export class ProductDetailComponent implements OnInit {
        constructor(private route: ActivatedRoute) {}

        ngOnInit(): void {
           this.route.data.subscribe(data => {
                this.product = data['product'];
           });
        }
}
```

Child routes

- With a child route <router-outlet> can be nested
- Can be used to easyly change the basic layout

```
const routes: Routes = [
{
  path: '',
  component: StandardLayoutComponent,
  children: [
   { path: '', component: WelcomeComponent },
   { path: 'employees', loadChildren:
        './employee/employee.module#EmployeeModule' },
   { path: 'about', loadChildren:
        './about/about.module#AboutModule' }
  ]
}];
```


Guards

- Guards can be helpful for:
  - limiting access to a route
  - warning before leaving a route
  - Retrieving data before accessing a route
- Guards processing:
  - canDeactivate
  - canLoad
  - canActivateChild
  - canActivate
  - resolve

Guards

- A simple class which implements an interface

```
@Injectable()
export class AuthGuard implements CanActivate {
    constructor(private authService: AuthService,
                private router: Router) { }

    canActivate(route: ActivatedRouteSnapshot, state:
                RouterStateSnapshot): boolean {
        return isLoggedIn;
    }
}
```


Guards - CanDeactivate

- Another example:

```
@Injectable()
export class EmployeeEditGuard implements
                CanDeactivate<EmployeeComponent> {

  canDeactivate(component: EmployeeComponent): boolean {
    return confirm(`Navigate away and lose all changes?`);
  }

}
```

Guards - CanDeactivate

- Guards need to be registered in our ngModule or in our project in our barrel

```
@NgModule({
  imports: [
    SharedModule,
    EmployeeRoutingModule
  ],
  exports: [],
  declarations: [...fromContainers.containers,
                 ...fromComponents.components, ...fromPipes.pipes],
  providers: [...fromServices.services, ...fromGuards.guards]
})
export class EmployeeModule {}
```


Guards - CanDeactivate

- Another example:

```
const routes: Routes = [
   {
     path: ':employeeId',
     component: fromContainer.EmployeeComponent,
     canActivate: [fromGuards.AuthGuard],
     canDeactivate: [fromGuards.EmployeeEditGuard]
   }
];
```

Preloading Feature Module

- Our feature module is configured for lazy loading
- Sometimes it is useful to preload it behind the scene.
- This can be done with Preloading (Eager Lazy Loading)
- There are three possibilites:
  - No preloading
  - Preload all
  - Custom (own preload strategy)

Preloading all

- To enable we pass a preload strategy to the forRoot() method
- CanLoad Guard can block preloading

```
@NgModule({
  imports: [RouterModule.forRoot(routes,
        { preloadingStrategy: PreloadAllModules })
  ],
  exports: [RouterModule],
  providers: []
})
export class AppRoutingModule {}
```

Custom Preloading Strategy

- This allows us that just some of the modules are preloaded
- A custom preloader is just a simple Angular service

```
export class AppCustomPreloader implements PreloadingStrategy {

  preload(route: Route, load: Function): Observable<any> {
    return route.data && route.data.preload ? load() : of(null);
  }

}
```


Custom Preloading Strategy

- Register our custom preloader to the forRoot() method

```
@NgModule({
  imports: [RouterModule.forRoot(routes,
        { preloadingStrategy: AppCustomPreloader })
  ],
  exports: [RouterModule],
  providers: [AppCustomPreloader]
})
export class AppRoutingModule {}
```

Custom Preloading Strategy

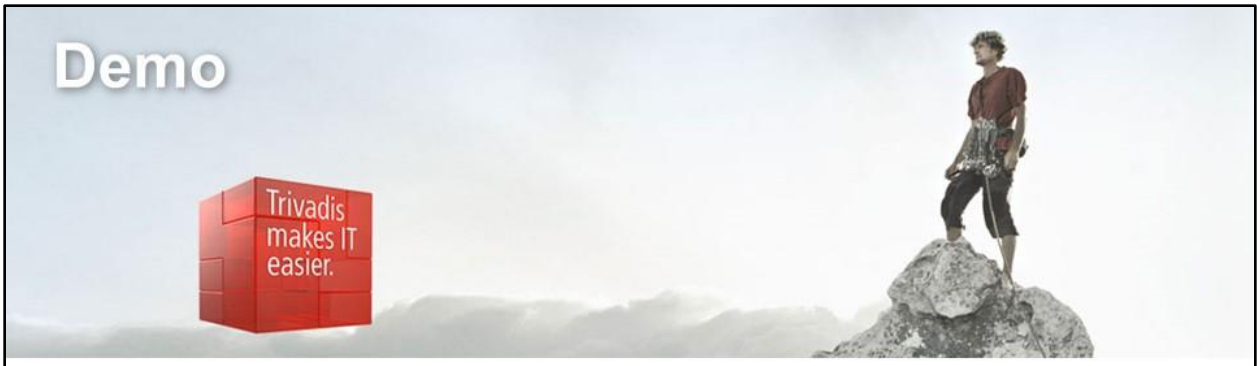
- And now mark the module which has to be preloaded

```
const routes: Routes = [{
    path: '', component: StandardLayoutComponent,
    children: [
      { path: '', component: WelcomeComponent },
      { path: 'employees', loadChildren:
               './employee/employee.module#EmployeeModule',
        data: { preload: true }
      },
       …
    ]
}];
```


Summary

- Angular has an integrated Router
- Routing means navigating to different components
- Providing data with a resolver
- Guards can be helfpul
- With a Custom Preloading Strategy we can eager lazy load some modules

Demo



Make sure lazy loaded is implemented

Add Query Parameter for search functionality

Create a Route Guard (e.g. when leaving our edit form and ask the user for it)

Create a custom preloading strategy

trivadis
makes IT easier.

20_TemplateDrivenForms

Form technologies

- Template-driven Forms
  - Use a Component's Template
  - Unit Test against DOM
- Reactive Forms (also known as model-driven)
  - Use a Component's Template
  - Create a Form Model in TypeScript (must be in sync with the template)
  - Unit Test against Form Model
  - Validation in Form Model

trivadis
makes IT easier.

Template-driven Forms

- Kind of forms used with Angular 1.x
- Bind directives and behavious to your templates
- Used with ngModel, required, minlength etc.
- The template is doing the work

Starting Position

```
<form novalidate> <!- only for Angular<4 -->
    <div class="form-group row">
        <label>Firstname</label>
        <input type="text" class="form-control">
    </div>
    <div class="form-group row">
        <label>Lastname</label>
        <input type="text" class="form-control">
    </div>
    <div class="form-group row">
        <label>Email</label>
        <input type="text" class="form-control">
    </div>
    <div class="form-group row">
        <label>Confirm Email</label>
        <input type="text" class="form-control">
    </div>
    <button type="submit"> Save </button>
</form>
```

Template-driven Forms

- Add the FormsModule to your AppModule to use ngModel

```
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 { }
```


Template-driven Forms

- Or better put FormsModule to your SharedModule

```
import { NgModule }      from '@angular/core';
import { CommonModule } from '@angular/common';
import {FormsModule} from '@angular/forms';

@NgModule({
  imports: [CommonModule],
  exports: [CommonModule, HttpClientModule, FormsModule],
  declarations: []
})
export class SharedModule { }
```

Template-driven Forms

- Declare our model class

```
@Component({...
  templateUrl: 'employee-form.component.html'
})
export class EmployeeFormComponent {
 employee:Employee = {firstname: 'Thomas',
               lastname: 'Gassmann',
               email: 'thomas.gassmann@trivadis.com'};
}
```


Template-driven Forms

- Binding ngForm and ngModel

```
<form novalidate #f="ngForm">
   <div class="form-group row">
        <label>Firstname</label>
        <input type="text"
               name="firstname"
               [(ngModel)]="employee.firstname"
               class="form-control">
    </div>
</form>
```

Template-driven Forms

- Add submit functionality by adding the ngSubmit event directive

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
   …
</form>
```

```
export class AppComponent {
 employee:Employee= {…};
 onSubmit({ valid }: { valid: boolean }) {
    console.log(valid)
 }
}
```


Template-driven error validation

- Similar approach as in Angular 1.x.
- Let's start by disabling our submit button until the form is valid

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
   …
   <button type="submit" [disabled]="f.invalid">Save</button>
</form>
```

Template-driven error validation

- Mark each <input> as required and show error message if needed

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <label>
   <span>Firstname</span>
   <input type="text" name="firstname"
        [(ngModel)]= "employee.firstname" required>
  </label>
  <div *ngIf="f.controls.firstname?.errors?.required">
       Firstname is required
  </div>
   …
   <button type="submit" [disabled]="f.invalid">Save</button>
</form>
```


Template-driven error validation

- Extend our forms with nested objects and data

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <label>
   <span>Firstname</span>
   <input type="text" name="firstname" #firstname="ngModel"
        [(ngModel)]= "employee.firstname" required>
  </label>
  <div *ngIf="firstname.errors?.required && firstname.touched"
       class="error">
       Firstname is required
  </div>
   …
   <button type="submit" [disabled]="f.invalid">Save</button>
</form>
```

Template-driven custom validation

- A custom validator is just a directive

```
@Directive({
  selector: '[appIsEmails][ngModel]',
  providers: [
    { provide: NG_VALIDATORS, useExisting: IsEmailValidator, multi:
true }
  ]
})
export class IsEmailValidator implements Validator {
  validate(c: AbstractControl) {
      return EmployeeValidators.emailValidator(c);
  }
}
```


Template-driven custom validation

```
export class EmployeeValidators {
  static emailValidator(control: AbstractControl):
                                        ValidationErrors | null {
    const val: string = control.value;
    if (!val || val.indexOf('@') > 0) {
      return null;
    }
    return { invalidemail: true };
  }
}
```

Template-driven custom validation

- Use it in markup

```
…
 <div class="form-group row">
        <label>Email</label>
        <input type="text" class="form-control"
               name="email" #email="ngModel"
               [(ngModel)]="employee.email" appIsEmails>

     <div *ngIf="email.errors?.invalidemail && email.touched">
            Invalid email
        </div>
 </div>
```


Summary

- Bind directives and behavious to our templates
- Used with ngModel, required, minlength etc.
- Template is doing the work

21_ReactiveForms

Reactive Forms

- Also known as model-driven
- Avoiding directives such as ngModel, required etc.
- Use the underlying APIs to do the work by creating a instance inside a component class and construct JavaScript models.
- Very testable
- Keeps all logic in the same place

trivadis
makes IT easier.

Starting Position

```
<form novalidate> <!- only for Angular<4 -->
    <div class="form-group row">
        <label>Firstname</label>
        <input type="text" class="form-control">
    </div>
    <div class="form-group row">
        <label>Lastname</label>
        <input type="text" class="form-control">
    </div>
    <div class="form-group row">
        <label>Email</label>
        <input type="text" class="form-control">
    </div>
    <div class="form-group row">
        <label>Confirm Email</label>
        <input type="text" class="form-control">
    </div>
    <button type="submit"> Save </button>
</form>
```


Reactive Forms

- Add the **ReactiveFormsModule** to your AppModule

```
…
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [CommonModule],
  exports: [CommonModule, HttpClientModule, ReactiveFormsModule],
  declarations: []
})
export class SharedModule { }
```

Reactive Forms – FormControl

- FormControl
  - A class that powers an individual form control
  - Tracks the value
  - Validation status

```
ngOnInit() {
  this.myNameCtrl = new FormControl('Thomas Gassmann');
}
```


Reactive Forms – FormGroup

- FormGroup
  - A group of FormControl instances
  - Tracks the value
  - Validation status
  - Can be nested (FormGroup in FormGroup)

```
ngOnInit() {
  this.myGroup = new FormGroup({
    name: new FormControl('tga'),
    email: new FormControl('thomas.gassmann@trivadis.com')
  });
}
```

But how we use them?

Reactive Forms – use of FormGroup

- For binding declare **formGroup** on the form and **formControlName** as a directive
  Note: ngModel and name attribute are no longer required

```
<form novalidate [formGroup]="employee">
  <label>
   <span>Firstname</span>
   <input type="text" formControlName="firstname">
  </label>
  …
</form>
```


Reactive Forms – implementing the FormGroup model

```
import { Component, OnInit } from '@angular/core';
import {FormControl, FormGroup} from '@angular/forms';

@Component({...})
export class EmployeeFormComponent {
 employee:FormGroup;
  ngOnInit() {
    this.employee = new FormGroup({
      firstname: new FormControl(''),
      lastname: new FormControl(''),
      email: new FormControl(''),
    });
  }
}
```

Reactive Forms – set form value

- To set a value from our model we can use setValue or patchValue
- With setValue all controls need to be set whereas patchValue can be used to set only some of the form values.

```
this.employee.setValue({ firstname: 'Thomas',
            lastname: 'Gassmann', email: 'tg' });

// OR


this.form.patchValue({ firstname: 'Thomas' });
```


Reactive Forms

- Exactly the same as the template-driven approach. Add submit functionality by adding ngSubmit event directive

```
<form (ngSubmit)="onSubmit()" [formGroup]="employee">
   …
</form>
```

```
export class EmployeeFormComponent {
 employee:FormGroup;
 onSubmit() {
    console.log(this.employee.valid)
 }
}
```

Reactive Forms error validation

```
import { Component, OnInit } from '@angular/core';
import {FormControl,FormGroup,Validators} from '@angular/forms';

@Component({...})
export class EmployeeFormComponent {
 employee:FormGroup;
  ngOnInit() {
    this.employee = new FormGroup({
      firstname: new FormControl('', Validators.required),
      email: new FormControl('',
                [Validators.required, Validators.minLength(2)]),
    });
  }
}
```


Reactive Forms error validation

- Let's start by disabling our submit button until the form is valid

```
<form novalidate (ngSubmit)="onSubmit()" [formGroup]="employee">
   …
   <button type="submit"
                [disabled]="employee.invalid">Save</button>
</form>
```

Reactive Forms error validation

- Use the .controls property on the FormGroup object to check for errors

```
<div *ngIf="employee.controls.firstname?.errors?.required">
       Firstname is required
</div>
```

- Or use .get() method that will lookup for that control

```
<div *ngIf="employee.get('firstname').hasError('required')
         && employee.get('firstname').touched">
       Firstname is required
</div>
```


Reactive Forms - FormBuilder

- Usage of FormGroup and FormControl is complex
- Let's simplify it with FormBuilder
- **FormBuilder** is a helper class to build forms
- It is a flexible and common way to configure forms

So, let's change our existing form

Reactive Forms - FormBuilder

- FormBuilder is injected into constructor

```
import { Component, OnInit } from '@angular/core';
import {FormBuilder,FormGroup,Validators} from '@angular/forms';

@Component({...})
export class EmployeeFormComponent {
 employee:FormGroup;
 constructor(private fb: FormBuilder) {}
  ...
}
```


Reactive Forms - FormBuilder

```
@Component({...})
export class EmployeeFormComponent {
 employee:FormGroup;

 constructor(private fb: FormBuilder) {}

 ngOnInit() {
    this.employee = this.fb.group({
      firstname: ['', Validators.required],
      email: ['',[Validators.required,Validators.minLength(4)]],
    });
  }
}
```

Reactive Forms – Read values from FromGroup

```
@Component({...})
export class EmployeeFormComponent {
 form:FormGroup;

 onSubmit() {
   if (this.form.touched && this.form.valid) {
     const newModel = Object.assign({}, this.model,
                                    this.form.value);
     const newModel2 = { ...this.model, ...this.form.value };

     this.update.emit(newModel);
    }
  }
}
```


Reactive Forms – Custom validation

- Just a simple method

```
export class EmployeeValidators {
  static emailValidator(control:
                        AbstractControl):ValidationErrors | null {
    const val: string = control.value;
    if (!val || val.indexOf('@') > 0) {
      return null;
    }
    return { invalidemail: true };
  }
}
```

Reactive Forms – Custom validation

- synchronous and asynchronous validations are possible

```
this.fb.group(
    {
        …
        email: [
           '',
            [Validators.required,
            EmployeeValidators.emailValidator],
            EmployeeValidators.checkEmailUniqueAsync(this.service)
        ]
});
```


trivadis makes IT easier.

updateOn option

- For Forms with heavy validation requirements, updating on every keystroke can sometimes be too expensive
- Angular 5 provides a new option that improves performance by delaying form control updates until the blur or the submit event.

```
email: new FormControl(null, {
        validators: Validators.required,
        updateOn: 'blur'
}),
```

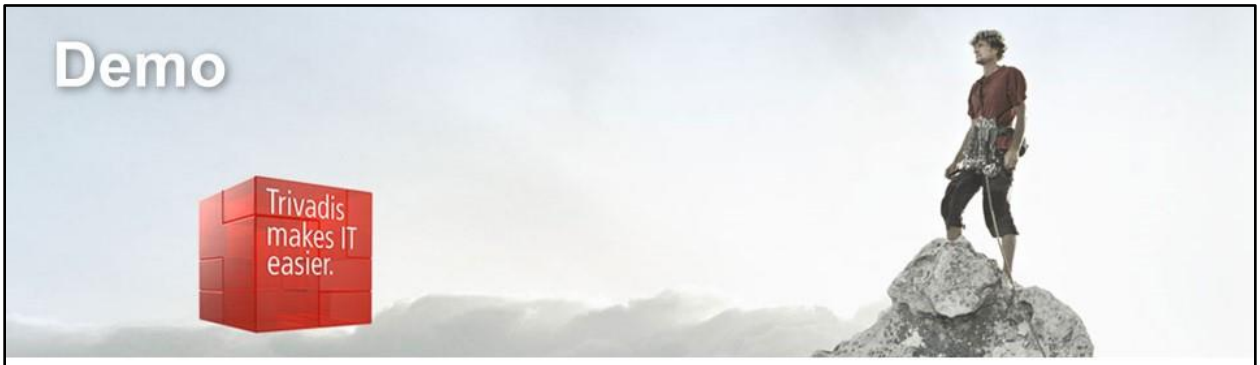
- **Not working with FormBuilder. Open bug!**

trivadis makes IT easier.

Summary

- Reactive-Forms are super powerful
- Avoid directives such as ngModel, required etc.
- Import ReactiveFormsModule

trivadis makes IT easier.



Create a employee form with reactive forms

trivadis makes IT easier.

22_UnitTesting

BASLE • BERN • BRUGG • DÜSSELDORF • FRANKFURT A.M. • FREIBURG I.BR. • GENEVA
HAMBURG • COPENHAGEN • LAUSANNE • MUNICH • STUTTGART • VIENNA • ZURICH

trivadis
makes IT easier.

What kind of tests exists?

Unit Tests

- Test certain functions or units of code
- Super fast
- Easy to write

e2e Tests

- Runs the real application and simulates the users behavior
- Very slow
- Very fragile

trivadis
makes IT easier.

Test Strategy

- Use the smallest test type possible
- Write readable tests
- Do not mock unless there is a reason to
- Test all the time
- Make tests fail

Tools

- Test runner
- Test code on various browser
- Integrate with various continuous integration services like Jenkins
- Complete configuration with Angular CLI
- *.spec.ts files are considered

```
ng test
```

```
ng test --code-coverage
```


- Behavior-driven development framework
- Fast
- Dependency free
- Already integreated with Angular CLI

Protractor

end to end testing for AngularJS

- End-to-end test framework
- Built on top of Selenium WebDriverJS
- Runs in a real browser
- Test like a user
- Protractor is also configured with the Angular CLI

```
ng e2e
```


Insiders' tip

Wallaby.js

- Integrated Continuous Testing Tool for JavaScript

Wallaby.js

```
21      it("is defined", () => {
22          console.log("Hallo Basta!");  Hallo Basta!
23          expect(comp).toBeTruthy();
24      });
```

```
21      it("is value equals", () => {
22          expect(1).toBe(2);  Expected 1 to be 2.
23      });
```

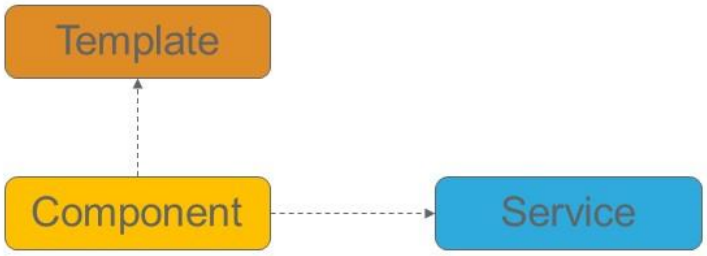

Unit Test vs. Integration Test

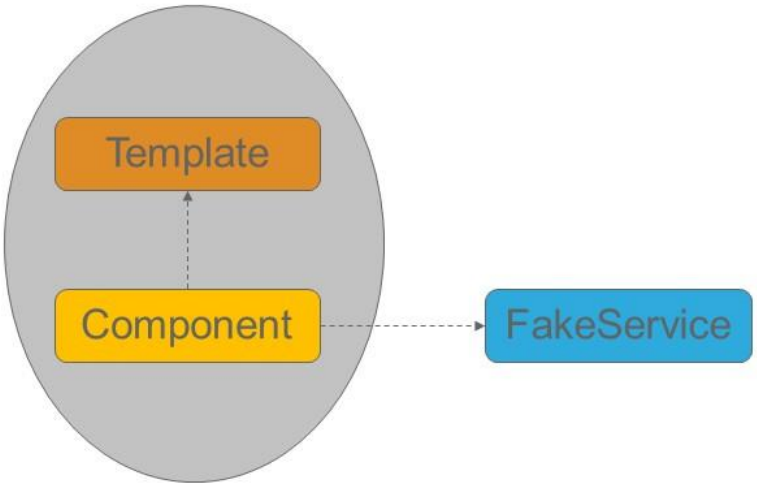
Unit Test

- Test in isolation, without external resources (e.g. templates)
- Just a simple function

Integration Test

- Test with external resources (e.g. templates)





Jasmin Basics

```
describe('Test Suite Name', () => {

        beforeAll(() => {});
        beforeEach(() => {});
        afterEach(() => {});
        afterAll(() => {});

        it('A Spect / Unit Test', () => {

        });

});
```


Jasmin Basics

```
describe('Test Suite', () => { // Test Suite
    it("A single spec", () => { // Spec or test
        // Arrange
        let component = new AppComponent();

        // Act
        component.ngOnInit();

        // Assert
        expect(component.title).toBe("hallo thomas");
    });
});
```

Test a pipe

- Pipe is only a class which implements an interface.

```
describe('FilterPipeTests', () => {
        let pipe: EmployeeFilterPipe;
        let employeees: Employee[];

        beforeEach(() => {
                pipe = new EmployeeFilterPipe();
        });

        it('should return all data', () => {
                const result = pipe.transform(employeees, null);
                expect(result.length).toBe(2);
        });
});
```


Test a Component

- Try to test with hardly any dependency to Angular
- When possible create a new instance directly

```
beforeEach(() => {
   comp = new PersonDetailComponent();
});
```

Test a Component

- Otherwise use the TestBed to configure a new TestingModule

```
beforeEach(
    async(() => {
      TestBed.configureTestingModule({
        imports: [RouterTestingModule, FormsModule],
        declarations: [EmployeeListComponent],
        schemas: [NO_ERRORS_SCHEMA]
      }).compileComponents();
    })
  );
```


Test a Component

- Fixture has a collection of helpful functions

```
beforeEach(() => {
    fixture = TestBed.createComponent(PersonDetailComponent);
    comp = fixture.componentInstance;
    fixture.detectChanges(); // triger change detection
});

 it('should render title in a h2 tag',
    async(() => {
      const result =
        fixture.nativeElement.querySelector('h2').textContent;
      expect(result).toContain('Employees');
    })
  );
```

Test a Component

- fixture.detectChanges triggers a change detection cycle
- Especially necessary when accessing bindings (e.g. {{ xyz }})
- whenStable() will be called when detectChanges is over

```
it('should render title in a h2 tag',
   async(() => {
       fixture.detectChanges(); // triger change detection
       fixture.whenStable().then(() => {
              expect(…)
       });
   })
 );
```


Test a Service

- Try to test with fewest possible dependencies to Angular
- We define our testing module and import HttpClientModule

```
beforeEach(() => {
   TestBed.configureTestingModule({
     imports: [HttpClientModule],
     providers: [EmployeeService]
   });
 });
 beforeEach(() => {
   service = TestBed.get(EmployeeService);
 });
```

Test a Service

- When using an async operation, always add the async function to your test
- Karma is not waiting for async operator
- A test without an assertion is always a passed test

```
it("should get a person async", async(() => {
    service.getPersonAsync(1).then((result) => {
        expect(result.firstname).toBe("Thomas");
    });
}));
```


FakeAsync

- The main advantage of fakeAsync over async is that the test appears to be synchronous.
- Calling tick() simulates the passing of time until all the pending asynchronous activities finish

```
it('should test fakeAsync',
  fakeAsync(() => {
      let flag = false;
      setTimeout(() => { flag = true; }, 50);
      expect(flag).toBe(false);
      tick(50);
      expect(flag).toBe(true);
  })
);
```

Mock a Service

- What if the test has no access to the REST Service
- We have to mock our service call

```
import { HttpClientTestingModule } from
'@angular/common/http/testing';
import { HttpTestingController } from
'@angular/common/http/testing';

 beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [EmployeeService]
    });
  });
```


Mock a Service

```
beforeEach(() => {
    service = TestBed.get(EmployeeService);
    httpMock = TestBed.get(HttpTestingController);
});
  it('should create the app', async(() => {
      const dummy: Employee[] = […];
      service.getEmployees().subscribe(list => {
         expect(list.length).toBe(2);
      });

    const request =
       httpMock.expectOne(`${environment.apiBaseUrl}/employees`);
      expect(request.request.method).toBe('GET');

      request.flush(dummy);
    })
  );
```

Use Firefox as a second test browser

- Install Firefox launcher for karma

```
npm install karma-firefox-launcher --save-dev
```

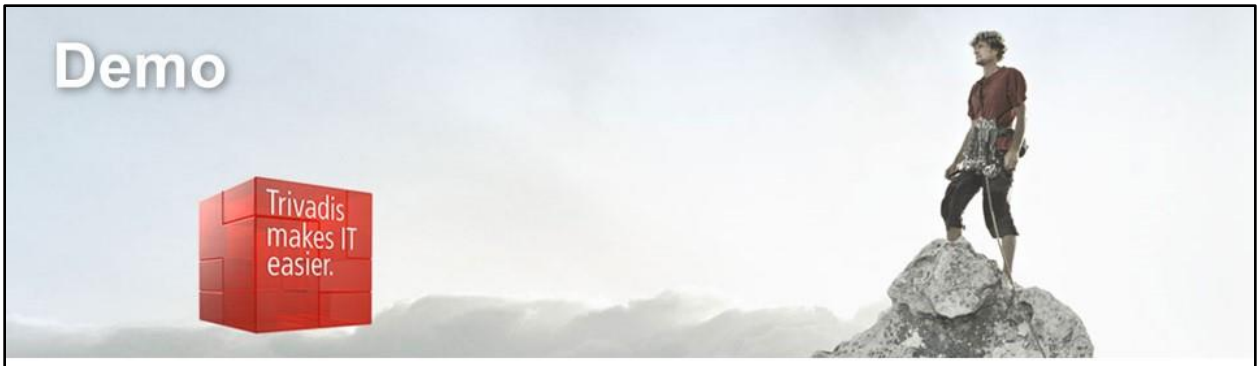
- Modify karma.conf.js

```
plugins: [ …
        require('karma-firefox-launcher'),
…
],
…
browsers: ['Chrome', 'Firefox'],
…
```


Summary

- Jasmin is the default testing framework. It works perfectly with all other tools
- Angular CLI configures Karma and Protractor
- Async functions should always be mapped inside an async keyword

Demo



Write an Unit-Test for employee-filter.pipe.ts

Write an Unit-Test “employee-list.component.ts”

Write an Unit-Test for employee.service.ts

Mock one of the connections

trivadis
makes IT easier.

23_e2eTesting

e2e Tests

- Test the entire app as a whole
- Simulate a real user
- With protractor

trivadis
makes IT easier.

Basics

```
import { browser, element, by } from 'protractor';

describe('QuickStart E2E Tests', function () {

    beforeEach(function () {
        browser.get('/');
    });

    it('should display title', function () {
        expect(element(by.css('h1')).getText()).toEqual("hallo basta");
    });

});
```


Basics

- "Page object"-classes are like ViewModels to interact with
- Each end-to-end test spec should have a "page object"-class
- Decapsulation from a specific framework like protractor
- Tests are located in a separate folder, e.g. "/e2e"

Page Object

```
import { AppPage } from './app.po';

describe('byod-portal App', () => {
    let page: AppPage;

    beforeEach(() => { page = new AppPage(); });

    it('should display welcome message', () => {
        page.navigateTo('/');
        expect(page.getText('app-root h2'))
            .toEqual('Byod Portal');
    });
});
```


Browser, Element, By

- The browser represents the WebDriver-Instance
- With by and element we can access HTML Elements

```
import { browser, by, element } from 'protractor';

export class AppPage {
    navigateTo(url: string) {
        return browser.get(url);
    }
    getText(selector: string) {
        return element(by.css(selector)).getText();
    }
}
```

Test navigation link

- Click simulates a real user click on an element

```
it('should navigate to byod page', () => {
    page.navigateTo('/');

    page.getElement('[ng-reflect-router-link="/employees"]')
        .click();

    expect(page.getCurrentUrl()).toContain('/employees');
});
```


Test navigation link – Page Object

- On the WebDriver-Instance (browser) the url can be read or set

```
getCurrentUrl() {
    return browser.getCurrentUrl();
}

…
```

Test the search field

```
it('should search correct', () => {
    page.navigateTo('/employees');

    page.search('Thomas');

    expect(page.getResult().count()).toEqual(3);
});
```


Test the search field – Page Object

- On an element, sendKeys can be used to send keys to an input

```
search(search: string) {
    this.getElement('input[type="search"]').clear();
    this.getElement('input[type="search"]').sendKeys(search);
}

getResult() {
    return element.all(by.css('table.table tbody tr'));
}
…
```

Other useful function

- Pause is for debugging during development

```
browser.pause();
```

- If the application is not an Angular App

```
browser.ignoreSynchronization = true
```

- Take a screenshot

```
browser.takeScreenshot().then(png => {
  const stream = fs.createWriteStream(`screenshot-1.png`);
  stream.write(new Buffer(png, 'base64'));
  stream.end();
});
```

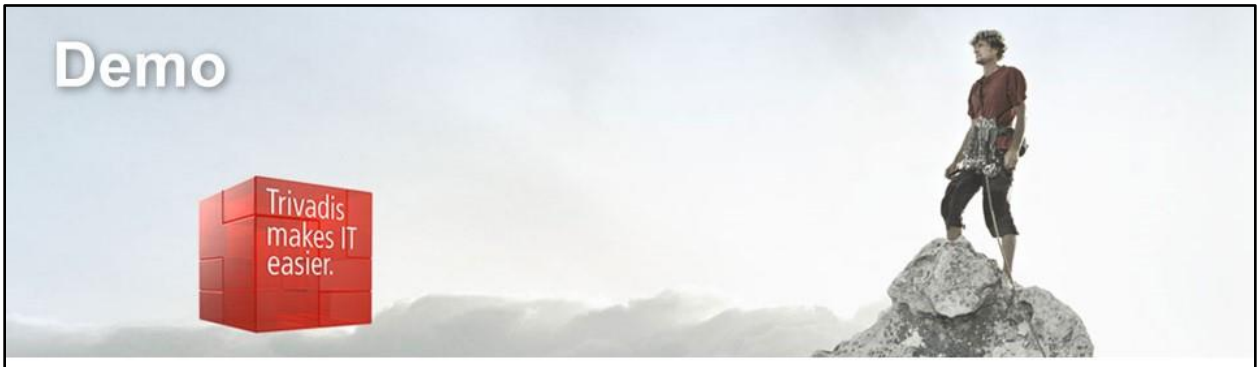

Settings

- Each time we execute end-to-end tests, an instance of chrome gets started, to powerup you can use the PhantomJS-plugin

Summary

- Protractor enables e2e tests
- browser, element and by are the most used commands for e2e tests
- e2e are an addition for unit testing.

trivadis makes IT easier.



Create an e2e test

Test the click on employee link

Test search functionality

trivadis makes IT easier.

24_Redux

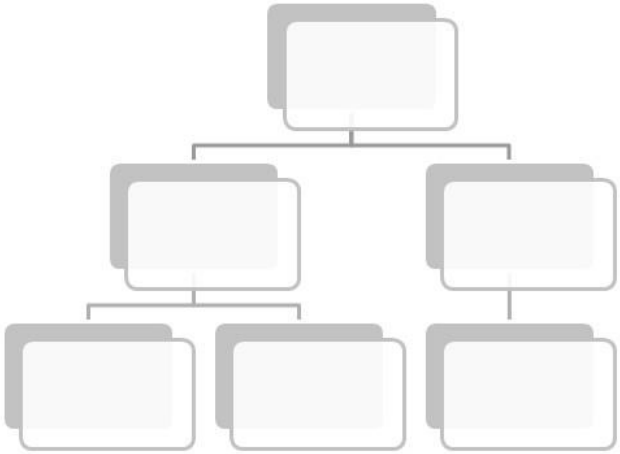
Redux

«Redux is not great for making simple things quickly. It's great for making really hard things simple.»

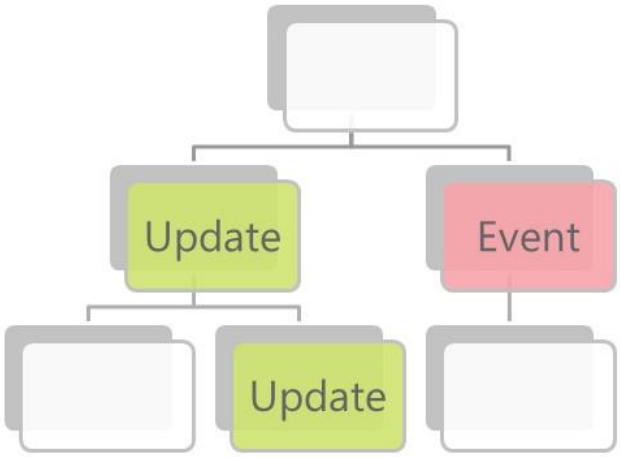
Jani Eväkal

trivadis
makes IT easier.

Typical Angular Application



Typical Angular Application



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

What is an Application State?

- Server response
- User input
- UI state
- Router / location state

The state is composed in our Store (and only in the store)

Redux Architecture: Core Concepts

- Single state tree
- Actions
- Reducers
- Store
- One-way dataflow

Redux: Single state tree

- One «big» JavaScript Object
- It is composed of reducers

```
// initial state
const state = {
        employees: []
};
```

Redux: Actions

- Actions have two properties: type and payload
- Dispatch actions to reducers

```
// an example of an action
const action = {
        type: 'ADD_EMPLOYEE',
        payload: {
                firstname: 'Thomas',
                lastname: 'Gassmann'
        }
};
```


Redux: Reducers

- Pure functions
- Given dispatched action
- Composes and returns new state

```
function reducer(state, action) {
    switch(action.type) {
        case 'ADD_EMPLOYEE': {
            const empl = action.playload;
            const newState = [...state.employees, empl];
            return { newState };
        }
    }
    return state;
}
```

Redux: Store

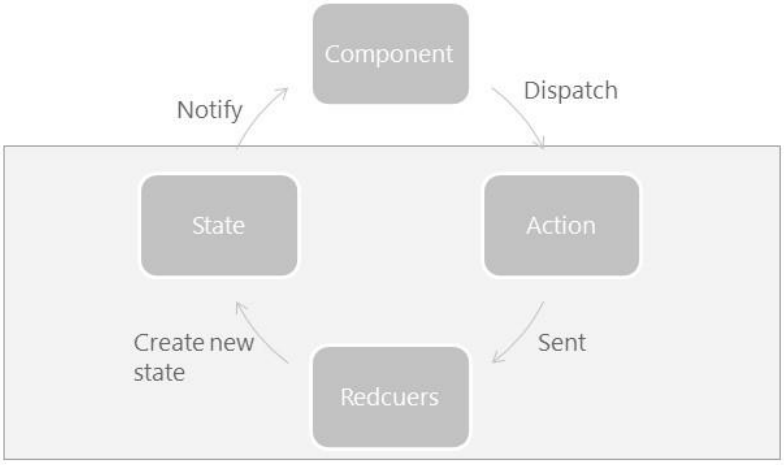
- State container
- Components subscribe to slices of state
- Components dispatch Actions to the store
- Reducers compose new State
- Store is updated and notifies subscribers
- Everything is immutable

Redux: Store

- Example of immutable operations

```
const person = {firstname: 'Thomas'};

const newP1 = Object.assign({}, person, {lastname: 'Gassmann'});
const newP2 = {…person, {lastname: 'Gassmann'});

console.log(person);  // {firstname: 'Thomas'}

console.log(newP1);  // {firstname: 'Thomas', lastname: 'Gassmann'}

console.log(newP2);  // {firstname: 'Thomas', lastname: 'Gassmann'}
```

One-way dataflow



trivadis makes IT easier.

Redux in Angular

- angular-redux

  https://github.com/angular-redux/store

- **ngrx**

  https://github.com/ngrx/platform

trivadis makes IT easier.

How to start

- Install npm package

```
npm install @ngrx/store --save
```

- Install effects module

```
npm install @ngrx/effects --save
```

- For debugging

```
npm install @ngrx/store-devtools --save
```

- Install «Redux» Chrome Extension

Integration

- Add references to your ngModule

```
@NgModule({
imports: [
        StoreModule.forRoot(fromStore.reducers),
        EffectsModule.forRoot(fromStore.effects),
        environment.production ? [] :
            StoreDevtoolsModule.instrument({
                maxAge: 10
            }),
],
…
})
export class AppModule {}
```

Create Actions

- Create a new File under actions/employee.action.ts
- Define action types according to what happen inside our app

```
export enum EmployeeActionTypes {
    LoadEmployees = '[Employees] Load Employee',
    LoadEmployeeSuccess = '[Employees] Load Employee Sucess',
    LoadEmployeeFail = '[Employees] Load Employee Fail'
}
```


Create Actions

- For loading all employees we have three different actions
- One for loading, one if it loads successful and one if there is an error

```
export class LoadEmployees implements Action {
  readonly type = EmployeeActionTypes.LoadEmployees;
}

export class LoadEmployeeSuccess implements Action {
  readonly type = EmployeeActionTypes.LoadEmployeeSuccess;
  constructor(public payload: Employee[]) {}
}

export class LoadEmployeeFail implements Action {
  readonly type = EmployeeActionTypes.LoadEmployeeFail;
  constructor(public payload: any) {}
}
```

Create Actions

- Export our actions for using it inside our reducer

```
// action types
export type EmployeeActions =
        | LoadEmployees
        | LoadEmployeeSuccess
        | LoadEmployeeFail
        ;
```


Create State Object

- Create a new File under state/app.state.ts
- State is just an interface and consists of other state objects

```
export interface AppState {
    employee: EmployeeState;
}
```

Create State Employee

- Create a new File under state/employee.state.ts
- File consists of the state and an initial state

```
export interface EmployeeState {
    employees: Employee[] ;
    loading: boolean;
    loaded: boolean;
}

export const initialState: EmployeeState = {
    employees : [],
    loading: false,
    loaded: false
};
```


Create Reducer

- Create a file reducers/employee.reducer.ts
- Reducer ist just a simple function which **always** returns a state

```
export function reducer(
  state = fromState.initialState,
  action: fromActions.EmployeeActions
): fromState.EmployeeState {

    switch(…) { …. }

    return state;
}
```

Implement our Reducer

- We have three actions at the moment. So let's implement the corresponding reducer
- LoadEmployee action

```
import * as fromActions from '../actions/employee.action';

switch (action.type) {
    case fromActions.EmployeeActionTypes.LoadEmployees: {
      return {
        ...state,
        loading: true,
      };
}
```


Implement our Reducer

- LoadEmployeeSuccess action
- Return always a new object (state).

```
case fromActions.EmployeeActionTypes.LoadEmployeeSuccess: {
      const employees = action.payload;

      return {
          ...state,
          loading: false,
          loaded: true,
          employees,
      };
  }
```

Implement our Reducer

- LoadEmployeeFail action
- Return always a new object (state).

```
case fromActions.EmployeeActionTypes.LoadEmployeeFail: {
      return {
          ...state,
          loading: false,
          loaded: false,
      };
 }
```


Define our Reducer Map

- Every NgRx Store always has at least one ActionReducerMap definition with the definition of all reducers inside this module
- This structure represents our state
- Create a new file reducers/index.ts

```
export const reducers: ActionReducerMap<fromState.AppState> = {
   employee: fromReducer.reducer
};
```

Use NgRx inside our component

- Inject the Store inside the constructor
- And select the appropriate projection

```
@Component({
 templateUrl: 'employee-list.component.html'
})
export class EmployeeListComponent implements OnInit {
   employees$: Observable<Employee[]>;
   constructor(private store$: Store<fromStore.AppState>) { }

   ngOnInit() {
       this.employees$ =
this.store$.select(fromStore.getEmployees);
   }
}
```


Create Selectors

- We make use of selectors to use a particular object of the state in our angular app.
- Inside our app state file. We export our employee state

```
export const getEmployeeState = (state: AppState) =>
state.employee;
```

- Now we create an export funtion for our array

```
export const getEmployees = (state: fromState.EmployeeState) =>
state.employees;
```

Create Selectors

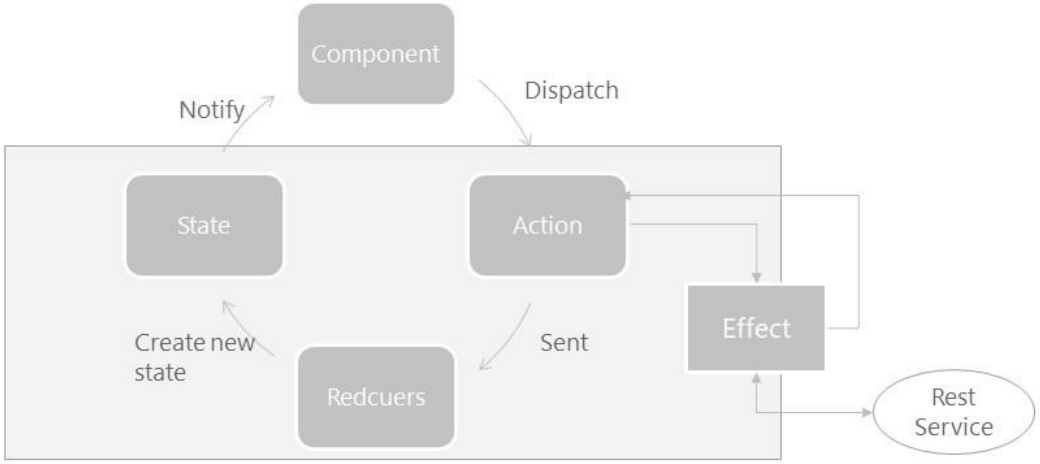
- We can now use these functions to create a selector. We put all our selectors inside a selector.employee.ts file

```
export const getAllEmployees = createSelector(
    getEmployeeState,
    fromReducer.getEmployees
);
```


Effects

- Effects are for side effects. When we access data from outside of our store we use effects
- Isolate side effects from components
- Effects are normal TypeScript classes with at least one function
- Functions have an «**@Effect()**» decorator

Effects



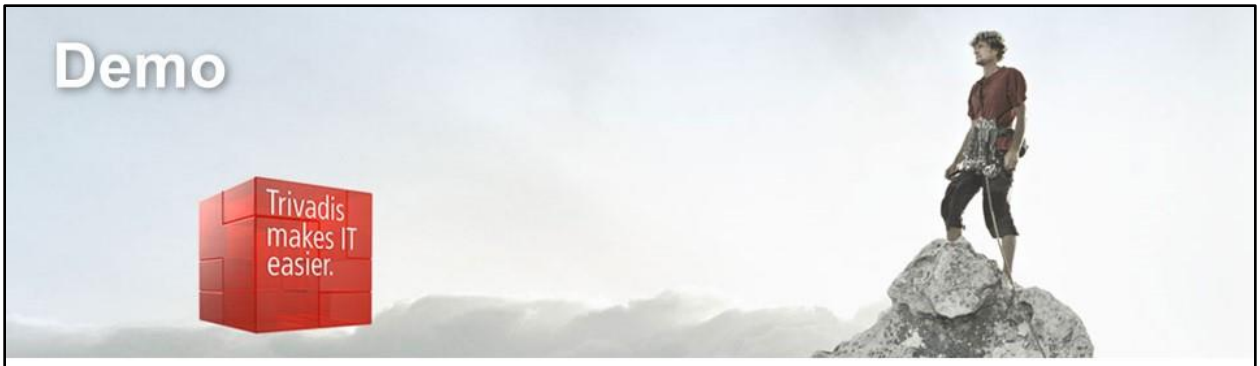
trivadis makes IT easier.

Create Effects

```
@Injectable()
export class EmployeeEffects {
  constructor(private actions$: Actions,
    private employeeService: fromServices.EmployeeService) {}

 @Effect()
  loadEmployees$ = this.actions$.ofType(
        employeeActions.EmployeeActionTypes.LoadEmployees)
   .pipe(
    switchMap(() => {
        return this.employeeService.getEmployees().pipe(
          map(employees => new
                          employeeActions.LoadEmployeeSuccess(employees)),
          catchError(error => of(new
                           employeeActions.LoadEmployeeFail(error)))
        );
    })
  );
}
```


makes IT easier.

Demo



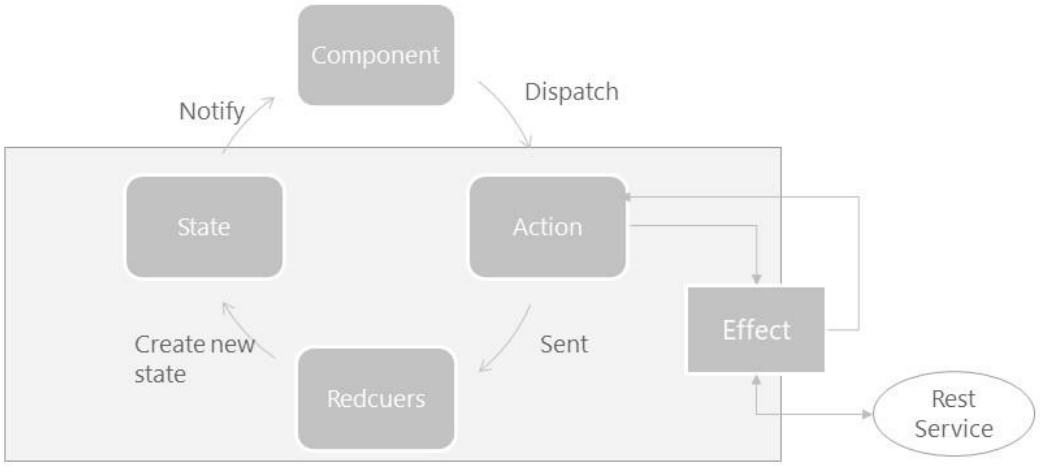
Add store functionality to our existing app

Modify our existing employee-list.component.ts to access our store

Create an effect to fetch the data

trivadis
makes IT easier.

Recap



trivadis
makes IT easier.

Advanced component architecture

Containers

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

Presentational

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

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 .

ChangeDetectionStrategy.OnPush

- Inside our component decorator

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


Demo

Summary

- NgRx is one of the most popular redux libraries in Angular
- It consists of Actions, Reducers and the Store
- Effects are for fetching data outside of our store

25_AngularElements

Angular Elements

«Angular is ideal for building **complete applications**, and our tooling, documentation and infrastructure are primarily aimed at this case.»

Rob Wormald, Angular Team

trivadis
makes IT easier.

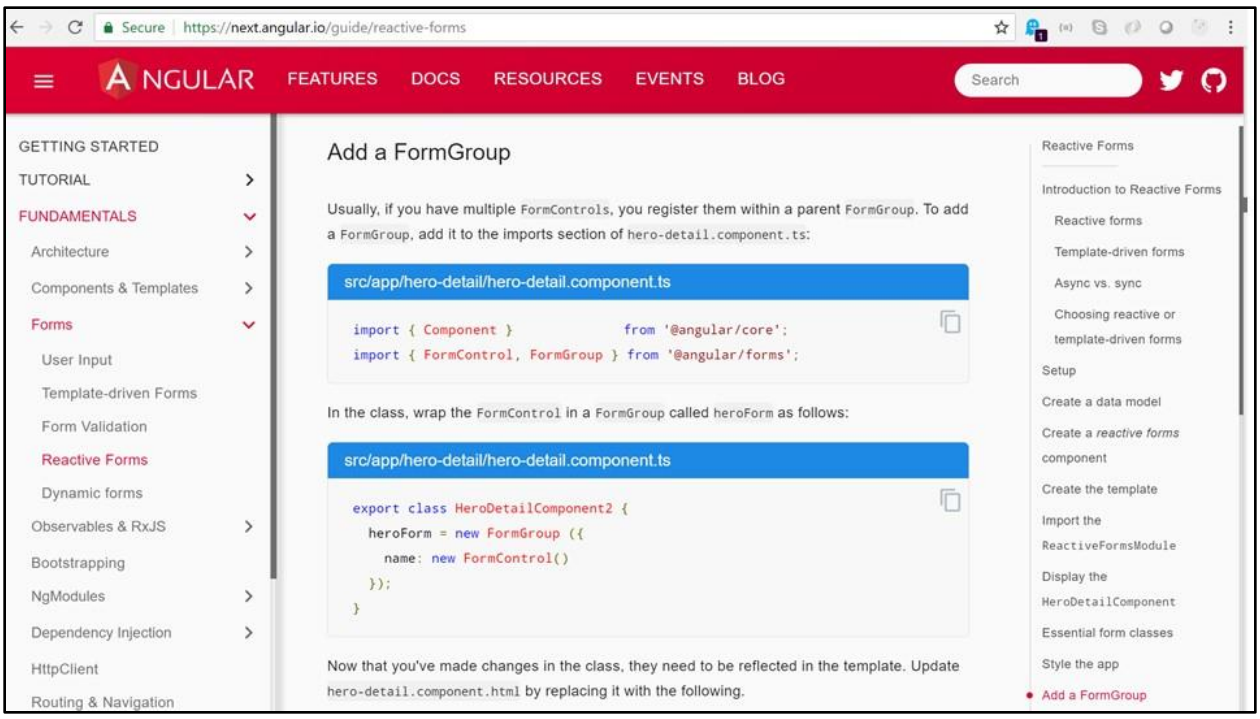
Angular Elements

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

Rob Wormald, Angular Team

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

Custom Elements

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

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

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 follow:

```
<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 follow:

```
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 follow:

```
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

Update Polyfil

- Add useful polyfil
  npm install @webcomponents/custom-elements –save
- Add polyfil 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 bootsrap 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 start module but do 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);
```

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 { … }
```


Build your component and pack it

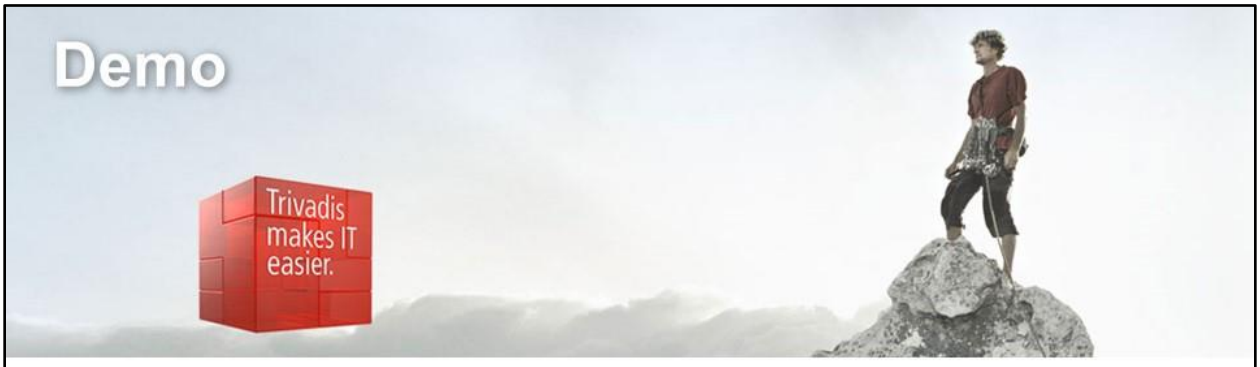
- Install concat and fs-extra
- Run following command

```
npm run build:elements
```

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

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.



Create an Angular Element component

Use it outside of Angular

26_Deployment

Steps to a successfull deployment

- TSLint
- Optimizing Code
- Build process

trivadis
makes IT easier.

TSLint

- Extensible static analysis tool
- Helps to catch common errors
- Catches the «oops» moments (debugger, console.log etc.)

Run TSLint

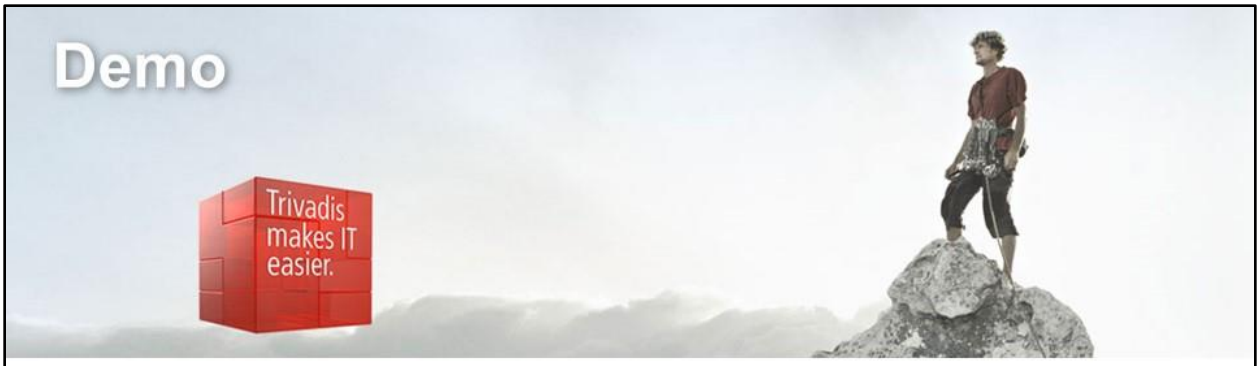
- Install TSLint

```
npm install tslint --save-dev          #local
npm install tslint -g                  #global
```

- Run TSLint

```
node_modules/.bin/tslint --init        #local: create tslint.json
tslint --init                          #global: create tslint.json
tslint "app/**/*.ts"                   #run TSLint
ng lint                                #using with Angular CLI
```

Demo



Integrate TSLint in your project

Check error and fix it

trivadis
makes IT easier.

Optimiziung RxJS (Angular <= 4)

- Often the whole module «rxjs/Rx» is imported

```
import "rxjs/Rx";
```

- affects the size of the bundle (webpack)
- and the amount of requests to the server (system.js)

trivadis
makes IT easier.

Optimiziung RxJS

- Use lettable operators
- Affects the size of the bundle (webpack)

```
this.service.update(employee).pipe(
        filter(newEmpl => newEmpl != null),
        tap(newEmpl => {
          // do something without returning a value
        }),
        map(newEmpl => new action.UpdateEmplSuccess(newEmpl)),
        catchError(error => {
          console.log(error);
          return of(new actions.UpdateEmplFail(error));
        })
      );
```


Imports RxJS

- New imports since rxjs v6

```
// creation and utility methods
import { Observable, Subject, pipe } from 'rxjs';

// operators all come from `rxjs/operators`
import { map, takeUntil, tap } from 'rxjs/operators';
```

- rxjs-compat package ensures backward-compatibility of RxJS.

```
npm install --save rxjs-compat
```

Build with Angular CLI

- Compile your app with the Angular CLI (JIT; since CLI 1.5 AOT default)

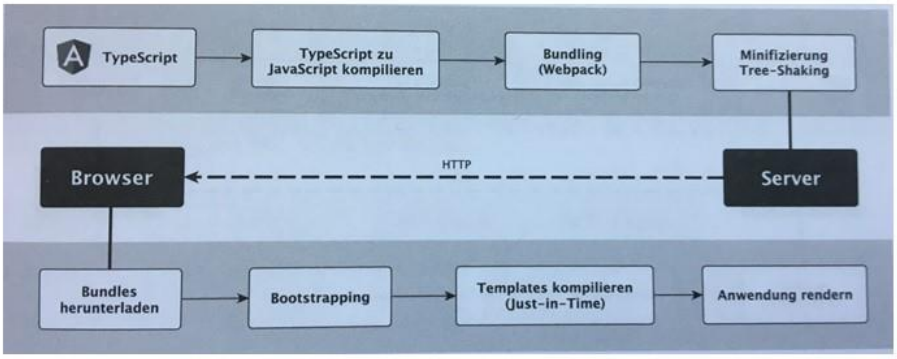
```
ng build                        #dev build
```

- Compile your app with the Angular CLI (AOT)

```
ng build --prod                 #prod build
```


Just in Time Compilation (JIT)

- Compilation is done during execution of our app



Woiwode et al. (2017)
https://angular-buch.com/

Angular in JIT Mode

```
// The browser platform with a compiler
import { platformBrowserDynamic }
       from '@angular/platform-browser-dynamic';


// The app module
import { AppModule } from './app.module';


// Compile (JIT) and launch the module
platformBrowserDynamic().bootstrapModule(AppModule);
```

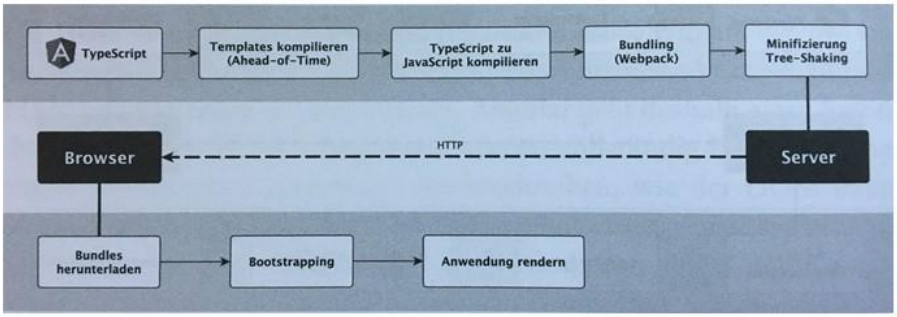

JIT Drawbacks

- Bundle size
- Slower rendering
- Errors in template can only be caught at runtime
- Security

The Angular 2 Compiler by Tobias Bosch (at Angular Connect 2016)
https://www.youtube.com/watch?v=kW9cJsvcsGo

Ahead of Time Compilation (AOT)

- Compilation is done during building our app



Woiwode et al. (2017)
https://angular-buch.com/

trivadis makes IT easier.

Angular in AoT Mode

```
// The browser platform with a compiler
import { platformBrowser } from '@angular/platform-browser';

// The generated app factory (AOT)
import { AppModuleNgFactory }
       from './aot/app.module.ngfactory';

// Launch with the app module factory.
platformBrowser().bootstrapModuleFactory(
   AppModuleNgFactory
);
```


trivadis makes IT easier.

Build for production

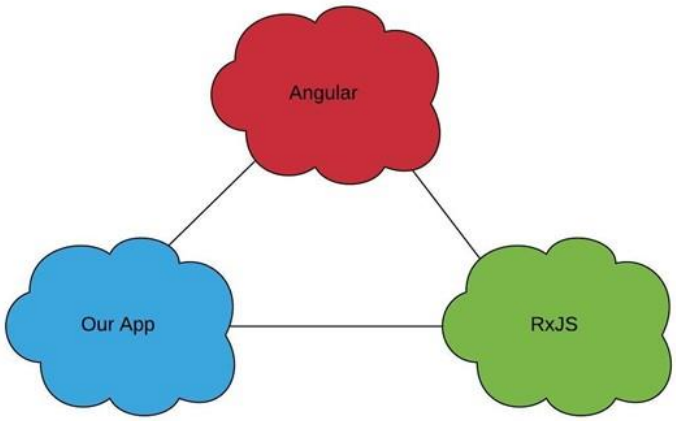
```
ng build --prod                    #prod build
```

- **AOT** compilation
- **Minification**: Comments, line breakings are removed and variables are shorten
- (limited) **Tree-Shaking**: Drop unused module export code
- **Hashing** of files
- **Build Optimizer**: limited dead code elimination via UglifyJS
- **Source Maps**: false

Minification

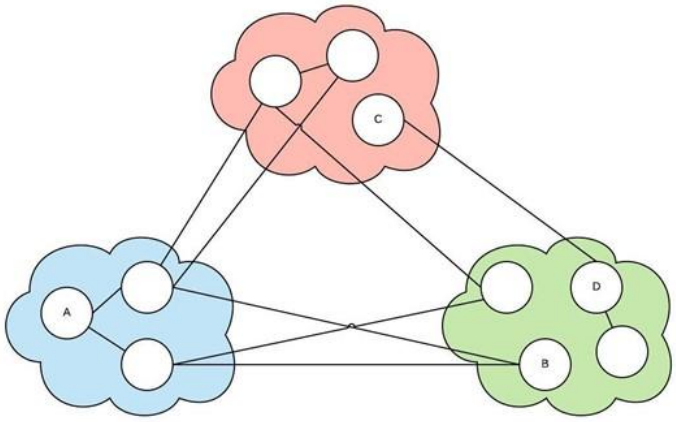
```
webpackJsonp([1,5],{"/fcW":function(l,n){function u(l){throw new Error("Cannot find module '"+l+"'.")}u.keys=function(){return[]},
u.resolve=u,l.exports=u,u.id="/fcW"},1:function(l,n,u){l.exports=u("x35b")},"1A80":function(l,n,u){"use strict";function t(l)
{return m._24(0,[(l()(),m._25(0,null,null,2,"a",[["class","navbar-brand"],["routerLink","/registrations"]],[[1,"target",0],[8,
"href",4]],[[null,"click"]],function(l,n,u){var t=!0;if("click"===n){t=!1!==m._26(l,1).onClick(u.button,u.ctrlKey,u.metaKey)&&t}
return t},null,null)),m._27(335872,null,0,g.y,[g.j,g.v,h.f],{routerLink:[0,"routerLink"]},null),(l()(),m._28(null,["Swiss
Microsoft CoWorking Space"]))],function(l,n){l(n,1,0,"/registrations")},function(l,n){l(n,0,0,m._26(n,1).target,m._26(n,1).href)})
}function e(l){return m._24(0,[(l()(),m._25(0,null,null,2,"a",[["class","navbar-brand"],["routerLink","/login"]],[[1,"target",0],
[8,"href",4]],[[null,"click"]],function(l,n,u){var t=!0;if("click"===n){t=!1!==m._26(l,1).onClick(u.button,u.ctrlKey,u.metaKey)&&
t}return t},null,null)),m._27(335872,null,0,g.y,[g.j,g.v,h.f],{routerLink:[0,"routerLink"]},null),(l()(),m._28(null,["Swiss
Microsoft CoWorking Space"]))],function(l,n){l(n,1,0,"/login")},function(l,n){l(n,0,0,m._26(n,1).target,m._26(n,1).href)})}
function i(l){return m._24(0,[(l()(),m._25(0,null,null,9,"div",[["style","color: white;\n        padding: 15px 50px 5px 50px;\n
    float: right;\n        font-size: 12px;"]],null,null,null,null,null)),(l()(),m._28(null,[" Last access:\n      "])),(l()(),
m._25(0,null,null,2,"time",[],null,null,null,null,null)),(l()(),m._28(null,["",""])),m._29(2),(l()(),m._28(null,["   "])),(l()(),
m._25(0,null,null,2,"a",[["class","btn btn-danger square-btn-adjust"],["routerLink","/logout"]],[[1,"target",0],[8,"href",4]],[
[null,"click"]],function(l,n,u){var t=!0,e=l.component;if("click"===n){t=!1!==m._26(l,7).onClick(u.button,u.ctrlKey,u.metaKey)&&t}
if("click"===n){t=!1!==e.authService.logout()&&t}return t},null,null)),m._27(335872,null,0,g.y,[g.j,g.v,h.f],{routerLink:[0,
"routerLink"]},null),(l()(),m._28(null,["Logout"])),(l()(),m._28(null,["\n    "]))],function(l,n){l(n,7,0,"/logout")},function(l,
n){var u=n.component;l(n,3,0,m._30(n,3,0,l(n,4,0,m._26(n.parent,0),u.authService.lastAccess,"dd.MM.yy HH:mm")))},l(n,6,0,m._26(n,7)
.target,m._26(n,7).href)})}function r(l){return m._24(0,[(l()(),m._25(0,null,null,8,"li",[],null,null,null,null,null)),(l()(),
```

Tree-Shaking



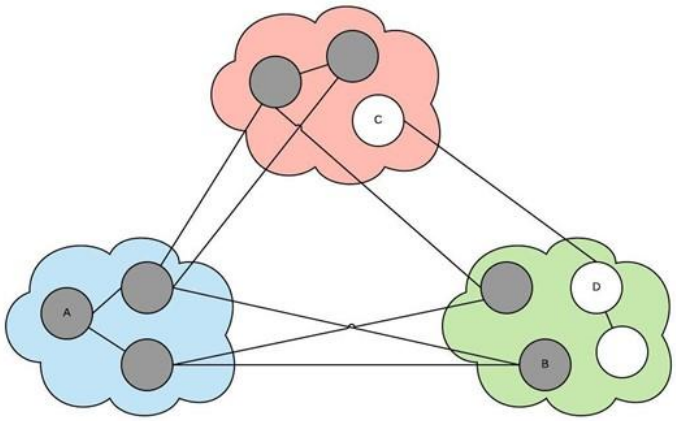
trivadis makes IT easier.

Tree-Shaking



trivadis makes IT easier.

Tree-Shaking



trivadis makes IT easier.

Bundles – Output of build

main.bundle.js	Source from our app
vendor.bundle.js	External libraries (Angular, RxJS)
inline.bundle.js	Logic for loading our bundles
styles.bundle.js	CSS Styles (only on DEV build)

trivadis makes IT easier.

Source Maps

- With source maps the size of the bundle can be analyzed.
- Install source-map-explorer

```
npm install -g source-map-explorer
```

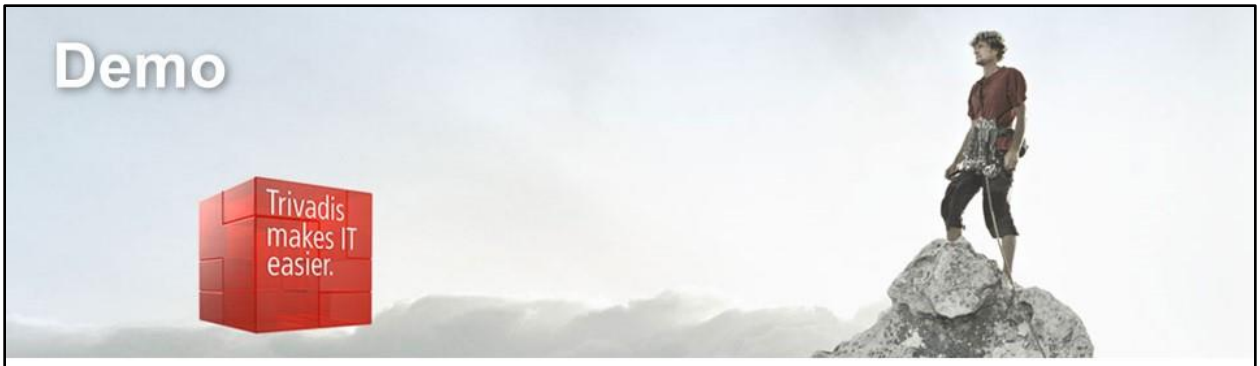
- Run viusalizer

```
source-map-explorer vendor.bundle.js
```


Summary

- Use TsLint during development
- When dealing with RxJs use new lettable operators (e.g. pipe)
- Angular CLI helps to create a dev or prod build
- Always use a prod build for production `ng build --prod`

Demo



Build your app

Play with setting

trivadis
makes IT easier.

27_WhatsNew

Versioning and releasing Angular

- In Angular 2 Semantic Versioning was introduced
- Every 6 month a new version
- Deprecation policy

trivadis
makes IT easier.

What's new in Angular 4

- Performance boost
- Angular Universal
- TypeScript's StrictNullChecks compliancy
- Stand alone animation module
- ngIf supports now else
- SEO optimizations
- httpClient (since v.4.3)
- …

What's new in Angular 4

```
<ng-template #hidden>
  <p>You are not allowed to see our secret</p>
</ng-template>
<p *ngIf="shown; else hidden">
  Our secret is being happy
</p>
```

What's new in Angular 4

```
//new in Angular 4 for SEO
this.title.setTitle(`TechEvent - ${this.pageTitle}`);
this.meta.addTag({
    name: "Description",
    content: `TechEvent - ${this.pageTitle}`
});
```


What's new in Angular 5

- Extended PWA Support
- Performance Boost
- Angular CLI: AoT default Build
- Forms *updateOn mechnism*
- I18N Pipes
- New Router events (ActivationStart, ActivationEnd, ChildActivationStart, ChildActivationEnd)
- …

What's new in Angular 6

- ng update
- ng add
- Angular Elements
- CLI Workspaces (workspaces containing multiple projects)
- Library Support (ng generate library <name>)
- Animations Performance Improvements
- ngModelChange is now emitted
- Tree Shakable Providers

Tree Shakable Providers

- No references are needed in our NgModule.

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

@Injectable({
  providedIn: 'root',
})
export class MyService {
  constructor() { }
}
```

Angular Labs

- *Schematics*
- *Component Dev Kit*
- ABC—The ABC (Angular + Bazel + Closure)
- *Angular Elements*

Summary

- Two major release per year
- Performance is getting better and better
- Angular CLI is one of the best starting points
- Schematics are super powerful

28_Course_Outro

Useful resources

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

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.

Feedback

https://mtm.cebglobal.com/url/u.aspx?412EAE43E132144804

https://tinyurl.com/yav3m5aq

trivadis makes IT easier.

Thank you

Thomas Gassmann
(@gassmannT)

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

trivadis
makes IT easier.