Angular 5 Tutorial

I use this document as a reference for Angular development. All code examples are copy & paste-able. Some code Examples work with complex Objects that were Parts of Programs I created or from Tutorials that I watched, so they need to be adapted to your specific use case. Also, I don't bother with removing all the css classes assigned to my HTML elements when I copy & paste my code in here.

Codevolution Angular 5 Tutorial - Youtube

Codevolution Angular 5 Tutorial - Github

Kudvenkat Angular CRUD Tutorial - Youtube

Blog: Angular in Depth

TO DO

How to customize the input date for Datepicker

Add links to the structural directive documentations and other ng directives

Add item from select to Array

Get the Prerequisites

Step 1: Get Node Package Manager (npm)

Download npm here.

Step 2: Get Angular

Installing Angular from npm is described here. This link also leads to the official documentation of Angular.

Angular CLI Commands

Configure CLI Output

Run `set DEBUG=express:*` to enable debug output

Run `set DEBUG=` to disable debug output

Generate a new App

Run `ng new app-name` to generate a new app in the current folder.

Run `ng serve --proxy-config proxy.conf.json` to launch the app and use the proxy route specified in proxy.conf.json

Launch the App

Run `ng serve` to launch the app.

Generate Components

Run `ng generate component component-name` to generate a new component.

Run `ng g c component-name` to generate a new component.

You can also use `ng generate directive/pipe/service/class/module`.

Notes:

- Components are created in a subdirectory be default - the subdirectory name is the component name. The filenames are .component.ts/html/css.
- Services and Pipes are *not* created/scaffolded in a subdir by default, see below.

Generate Services and Models

To create a service in a subdirectory do the following:

```
mkdir services
ng g s services\myservice
```

More examples:

```
# typical folder structure
mkdir services
mkdir models

ng g c models\user              # create a "User" model class
ng g s services\user            # create a "UserService" service
ng g s services\authentication  # create a "AuthenticationService" service
```

Building Blocks

Modules

Every Module represents a feature area in the Application

e.g. User Module, Admin Module

Root Module = app-module

Every Module is made up of one or more Components and Services

Components

Every Component controls a portion of the view - consisting of an HTML template and a TypeScript codebehind file that controls the logic of the view

e.g. Navigation Bar, Sidebar, Main Content

Root Component = app-component (tree structure - all other components are nested inside the root component)

The Component is declared by the Component Decorator, the 'templateUrl' and 'styleUrls' tell Angular where the HTML and CSS for this component are located. The 'selector' is the name to use in HTML Markup to declare this component.

```
@Component({
  selector: 'app-root',
```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

A Component can be used in different ways, depending on the declaration of the selector:

`selector: 'app-componentname'` will make the component available as an HTML element: `<app-componentname></app-componentname>`

`selector: '.app-componentname'` will make the component available as a class name: `<div class="app-componentname"></div>`

`selector: '[app-componentname]'` will make the component available as an attribute: `<div app-componentname"></div>`

Services

A Service is a class that contains the business logic of the Application

CSS

Class Binding

Single class:

```
<h1 class="myHeadline">Headline</h1>
```

Multiple classes:

```
<h1 class="myHeadline frontPage anotherClass">Headline</h1>
```

Fluent Class Binding

You can assign the class name to a property and bind the property to an html element using this syntax:

component.ts

```
public myClass = "myHeadline";
```

component.html

```
<h1 [class]="myClass">Headline</h1>
```

component.css

```
.myHeadline {
    color: red;
}
```

Class Binding turns regular class assignments into dummy assignments, so only the bound classes will apply when using mixed syntax

component.html

```
<h1 class="someClass" [class]="myClass">Headline</h1>
```

In the example above, only myClass will apply

Conditional Fluent Class Binding

Single class

There is an alternative syntax that binds one class based on a truthy or falsy value. If true, the CSS is applied.

component.ts

```
public useCSS = true;
```

component.html

```
<h1 [class.myHeadline]="useCSS">Headline</h1>
```

component.css

```
.myHeadline {
    color: blue;
}
```

Multiple classes = ngClass Directive

To conditionally apply multiple classes, use the ngClass Directive. Assign the booleans in the typescript file to the class declarations.

component.ts

```
public hasError = false;
public isSpecial = true;

public messageClasses = {
    "text-success": !this.hasError,
    "text-danger": this.hasError,
    "text-special": this.isSpecial,
}
```

component.html

```
<h2 [ngClass]="messageClasses">Some Text</h2>
```

component.css

```
.text-success {
    color: green;
}
.text-danger {
    color: red;
}
.text-special {
    font-style: italic;
}
```

Style Binding

Inline binding to a single style parameter works as follows:

```
<h2 [style.color]="'orange'">Text</h2>
```

Conditional Style Binding

Conditional Style Binding looks like this: if *hasError* is true, then Text is red. Else it is green.

component.ts

```
public hasError = false;
```

component.html

```
<h2 [style.color]="hasError ? 'red' : 'green'">Text</h2>
```

Fluent Style Binding

If two styles are not enough for the desired application, it is possible to bind an element to a string value that can be changed at runtime.

component.ts

```
public highlightColor = "orange";
```

component.html

```
<h2 [style.color]="highlightColor">Text</h2>
```

One Way Data Binding

Data Flow inside a Component: Class => View

Interpolation - works only with string values

Use `{{variable-name}}` to use the value of a variable in the HTML view

Can use `{{variable-name.member-name}}` (etc...) aswell

Can also use it with functions:

```
{{2+2}}
{{"Some String" + variable-name}}
{{string-variable-name.length}}
{{string-variable-name.toUpperCase}}
{{string-variable-name.myCustomFunction}}
```

Can *NOT* assign variables

```
{{a=2+2}}
```

Can *NOT* access global javascript variables

```
{{window.location.href}}
```

Use a variable in the component.ts for that:

component.ts

```
public siteUrl = window.location.href
```

component.html

```
{{siteUrl}}
```

Property Binding - can also work with boolean values

Attributes - Initial Values defined by HTML, read by using `(.getAttribute('value'))`

Attributes initialize Properties with predefined values and then they are done. Their values cannot change once they are inizialized.

Properties - Current Values defined by the DOM, read by using `(.value)`

Property values can change - they represent the current value.

component.ts

```
public myId = "testId";
```

component.html

```
<input [id]="myId" type="text" value="Vishwas">
```

binding a boolean Property like

component.html

```
<input [disabled]="true" type="text" value="Vishwas">
```

using a boolean Variable to make the view respond when the Variable value changes

component.ts

```
public isDisabled : boolean = true;
```

component.html

```
<input [disabled]="isDisabled" type="text" value="Vishwas">
```

component.html (alternative syntax)

```
<input bind-disabled="isDisabled" type="text" value="Vishwas">
```

Data Flow inside a Component: View => Class

To interact with the typescript class from the view, Angular offers Event Binding. Simply add the type of Event like "click" to the Element and assign the name of the function that should be called to it.

component.ts

```
private onClick() {
  console.log("onClick");
}
```

component.html

```
<button (click)="onClick()" type="button">Click Me</button>
```

Angular can also pass information about the event to the function

component.ts

```
private onClick(event) {
  console.log(event);
}
```

component.html

```
<button (click)="onClick($event)" type="button">Click Me</button>
```

Angular can also pass user inputs to the class

component.ts

```
logMessage(message) {
  console.log(message);
}
```

component.html

```
<input #myInput type="text">
<button (click)="logMessage(myInput.value)">Click Me</button>
```

Angular can also execute code manually right from the HTML

component.ts

```
public sometext = "";

private onClick() {
  console.log(sometext);
}
```

component.html

```
<button (click)="sometext='this text was changed inline directly from HTML'"
type="button">Click Me</button>
<button (click)="onClick()" type="button">And me afterwards</button>
```

Pipes

Pipes allow to transform the data before showing it in the view. The data is *only* transformed for the view. The values of the properties in the class do *not* change

String Pipes

component.ts

```
name = "StringPipeTest";
title = "this is the title";
slice = "0123456789";
person = {
    "firstName": "John",
    "lastName": "Doe"
}
```

component.html

display the string in all lowercase or all uppercase

```
<h2>{{ name | lowercase }}</h2>
<h2>{{ name | uppercase }}</h2>
```

display the string with first letter of every word capitalized

```
<h2>{{ title | titlecase }}</h2>
```

display part of the string, starting with (zero-based) index 3

```
<h2>{{ slice | slice:3 }}</h2>
```

display part of the string, starting with (zero-based) index 3 up to index 4

```
<h2>{{ slice | slice:3:5 }}</h2>
```

display the string in lowercase

```
<h2>{{ person | json }}</h2>
```

Number Pipes

component.html

number pipe specifies the number of digits and decimal digits: '1.2-3' = shows 5.678 - minimum number of digits = 1 - decimal digits minimum 2, maximum 3

```
<h2>{{ 5.678 | number:'1.2-3' }}</h2>
```

shows 05.6780

```
<h2>{{ 5.678 | number:'2.4-5' }}</h2>
```

shows 005.68

```
<h2>{{ 5.678 | number:'3.1-2' }}</h2>
```

turns a decimal into its percentage => shows 25%

```
<h2>{{ 0.25 | percent }}</h2>
```

no currency code: default shows $0.25 (US Dollars)

```
<h2>{{ 0.25 | currency }}</h2>
```

currency code 'EUR': shows €0.25 (Euro) check the ISO Currency Code List for more currency codes

```
<h2>{{ 0.25 | currency: 'EUR' }}</h2>
                                                //
<h2>{{ 0.25 | currency: 'EUR' : code }}</h2>    //  shows the code instead of the currency
sign: shows EUR0.25
```

Date & Time Pipes

component.ts

```
date = new Date();
```

component.html

```
<h2>{{date}}</h2>
<h2>{{date | date:'short'}}</h2>
<h2>{{date | date:'shortDate'}}</h2>
<h2>{{date | date:'shortTime'}}</h2>
<h2>{{date | date:'medium'}}</h2>
<h2>{{date | date:'mediumDate'}}</h2>
```

```
<h2>{{date | date:'mediumTime'}}</h2>
<h2>{{date | date:'long'}}</h2>
<h2>{{date | date:'longDate'}}</h2>
<h2>{{date | date:'longTime'}}</h2>
```

Two Way Data Binding

Data Flow inside a Component: View <=> Class

When working with input elements, it is essential that the model (in the class) and the user input (in the view) are always in sync. For this purpose, Angular provides the ngModel directive.

The basic syntax is a Mix between [Propery Binding] and (Event Binding):

component.ts

```
public name = "";
```

component.html

```
<input [(ngModel)]="name" type="text>
{{name}}
```

To use the ngModel directive, we need to import the FormsModule from '@angular/forms'

app.module.ts

```
...

import { FormsModule } from '@angular/forms';

...


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

Two Way Binding a Boolean Value to two Radio Buttons

To bind two Radio Buttons to a Model that has a boolean value, these steps must be taken:

1.) Give both inputs the same `name` attribute, so they can not be triggered at the same time.

2.) Bind the inputs to the same same property. In this case, its the *isPrivate* property of the *Insurance* object.

3.) The *value* has to be set with square braces (Property Binding) to "*true*" or "*false*"

component.html

```
<form #insurancesForm="ngForm">
    <div class="row dualRadio">
        <div>
            <input type="radio" name="private" [(ngModel)]="Insurance.isPrivate"
[value]="false"><label>false</label>
        </div>
        <div>
            <input type="radio" name="private" [(ngModel)]="Insurance.isPrivate"
[value]="true"><label>true</label>
        </div>
    </div>
</form>
```

component.ts

```
Insurance = new Insurance();
```

Two Way Binding String Values to a Dropdown List

To bind the String Values selected in a Dropdown to a Model, these steps must be taken:

1.) Give both the label and the select the same `name` attribute, so they are linked.

2.) Use the ngModel directive as usual to bind the value of the dropdown to the Property. In this case, it is the `gender` Property of the `Patient` Object.

3.) The *value* has to be set with square braces (Property Binding) to "*true*" or "*false*"

component.html

```
<label for="gender">Gender</label>
<select id="gender" name="gender" [(ngModel)]="Patient.gender">
  <option *ngFor="let gender of genders">{{gender}}</option>
</select>
```

component.ts

```
Patient = new Patient();

private genders : string[] = [
    "male",
    "female",
```

```
    "not sure"
];
```

Two Way Binding DateTime Values to a DatePicker

It is bad practice to use the browser built-in DatePicker, because the implementation varies from browser to browser in terms of layout and data format. To achieve the same function, look and feel across all browsers, it is recommended to use a custom DatePicker. This example will be using the Datepicker from Valor Software's ngx-bootstrap

The Getting Started Guide provides Information about this, except for the part how to install bootstrap 3 / 4.

Step 1: Install ngx-bootstrap

```
npm install ngx-bootstrap --save
```

Step 2: Install bootstrap 3 or 4

```
npm install bootstrap@3 --save

npm install bootstrap@4 --save
```

Step 3.1: Either reference the bootstrap stylesheet in your *.angular-cli.json*

Find the "*styles*" section in your *.angular-cli.json* file and add the following line.

```
  "styles": [
    ...
    "../node_modules/bootstrap/dist/css/bootstrap.min.css",
    ...
  ],
```

Step 3.2: Or reference the bootstrap stylesheet in your *index.html*

Add one of these lines to the index.html of your application, depending on which version of bootstrap you installed.

index.html

```
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
rel="stylesheet">

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
rel="stylesheet">
```

Step 4: Import the desired component into the *app.modules.ts* and add it to the *imports* Array

In this example, we import the Datepicker. The Datepicker Documentation Page contains the import commands that have to be included.

app.module.ts

```
// RECOMMENDED (doesn't work with system.js)
import { BsDatepickerModule } from 'ngx-bootstrap/datepicker';

// or
import { BsDatepickerModule } from 'ngx-bootstrap';

@NgModule({
  imports: [BsDatepickerModule.forRoot(),...]
})
export class AppModule(){}
```

Step 5: Reference the Datepicker stylesheet in *.angular-cli.json*

Find the *"styles"* section in your *.angular-cli.json* file and add another line that points to the component you want to use.

```
  "styles": [
    ...
    "../node_modules/ngx-bootstrap/datepicker/bs-datepicker.css",
    ...
  ],
```

Step 6: Declare a Datepicker in your Application

Declare an *input* and set its type to "text" (NOT "date"!) and add the *bsDatepicker* directive to the element as shown in the example below. Then just use the *[(ngModel)]* directive to bind the Datepicker to the Model's Date value (e.g. Birthday).

component.html

```
<div class="input">
    <label>Birthday</label>
    <input [(ngModel)]="selectedPatient.birthday" name="birthday" #birthdayControl="ngModel"
bsDatepicker type="text">
</div>
```

component.ts

```
public selectedPatient : Patient = new Patient();
```

Step 7: Customize the Component

Check the Theme Reference of the Component to find a proper Theme. Then Import the Config Object into the Component that uses the Datepicker. Then create a Property to hold the values. Check the definition in *bs-datepicker.config.d.ts* (right-click => Go to Definition in VS Code) to find all the Properties it has. Use the Object.assign()

Method to assign the desired configuration to the Element. Then Property Bind the *datePickerConfig* Object to the bsConfig Property of the Datepicker in the HTML Tag.

component.ts

```
import { BsDatepickerConfig } from 'ngx-bootstrap/datepicker'

datePickerConfig: Partial<BsDatepickerConfig>;

constructor(private _patientService: PatientService) {
    this.datePickerConfig = Object.assign({}, { containerClass: 'theme-dark-blue',
showWeekNumbers: true});
}
```

component.html

```
<div class="input">
    <label>Birthday</label>
    <input [(ngModel)]="selectedPatient.birthday" name="birthday" #birthdayControl="ngModel"
bsDatepicker [bsConfig]="datePickerConfig" type="text">
</div>
```

Step X: Customize the Input Date Format [TBD]

I used the ngModel directive to bind the Datepicker to the Birthday Property of my Model. Now my Forms Model accepts the Value from the Datepicker just fine (View => Class). The Format looks like this:

```
"2018-04-19T08:35:25.000Z"
```

But when I want the Datepicker to be set from the Property of the Model (Class => View), it sais 'Invalid date'. Then the Format looks like this:

```
"2018-04-17T00:00:00+02:00"
```

This is of course not the desired behavior. The Datepicker be able to assign a value to the Patient Object, but it should also be able to show the initial value when the Patient is loaded and his / her Birthday has a different Format.

Two Way Binding a Complex Object from a Dropdown to another Object

The Data Types in this example are like a simple Web Shop would use them. An *Item* Object is bound to a *Customer* Object. To work with the *Item* Object-Oriented, it is insufficient to bind the ID of the Item (if it even has one...) or a string representation. The *Customer* has a Property e.g. MyItem, which is of Type Item, so the expected behavior is to bind an Item-Object to that Property. And this is the Syntax:

component.html

```
  <div class="form-group">
    <label for="singleItem">Dropdown for assigning a single Object</label>
    <select required id="singleItem"  class="form-control inputResize"
#singleItemControl="ngModel" name="singleItem" [(ngModel)]="selectedCustomer.singleItem">
      <option *ngFor="let item of items" [ngValue]="item">{{item.description}}</option>
```

```
    </select>
  </div>
```

The [ngValue] directive is the key here. It is bound directly to the "item" from the **ngFor* directive, so it points to the object itself. Using [value]="item" isn't going to do the trick, it is just going to bind a string to the Property *selectedCustomer.singleItem*.

Adding an Item to an Array from the View by selecting it from a Dropdown

Removing an Item from an Array from the View by clicking a Table Row

Pass the index of the list view to an "onDelete" Method and delete the Object from the Array from there.

component.html

```
<table class="listView" id="table">
    <tr>
      <th>Delete Example</th>
    </tr>
    <tr *ngFor="let patient of patients; index as i" title={{patient.patientId}}
(click)="onDeletePatient(i)">
      <td>{{patient.lastName}}, {{patient.firstName}}</td>
    </tr>
</table>
```

component.ts

```
onDeletePatients(i: number) {
    this.patients.splice(i, 1);
}
```

Forms in Angular

Most applications require some type of Form elements that the user can interact with to enter new data or change existing data. There are two different ways to create Forms in Angular. Template Driven Forms, which are used for simple Forms or Model Driven Forms (Reactive Forms), which allow for more complex Forms that use cross-field validation etc.

Template Driven Forms

Template Reference Variable

`#employeeForm` is called the Template Reference Variable. Notice `ngForm` was assigned as the value for the Template Reference Variable `employeeForm`. So `employeeForm` variable holds a reference to the form. When Angular sees a form tag, it automatically attaches the `ngForm` directive to it. The `ngForm` directive supplements the form element with additional features. It holds all the form controls that we create with `ngModel` directive and name attribute, and monitors their properties like value, dirty, touched, valid etc. The form also has all these properties.

Angular Form Directives

This is an example for a Template Driven Form. It uses the `ngModel` directive for Two-Way Databinding, although there is not underlying Object in the TypeScript class yet. The `ngForm` Directive also creates a dummy Object to hold the values in

this Form, so this Directive is predestined to use with Object Oriented Programming. The Object can be displayed in the view using Interpolation with *json* Pipe: `{{employeeForm.value | json}}`. The `ngSubmit` directive is called when the button with `type="submit"` is clicked and passes the whole Template Reference Variable to the `saveEmployee()` Method.

component.html

```
<form #employeeForm="ngForm" (ngSubmit)="saveEmployee(employeeForm)">
 <div class="panel panel-primary">
   <div class="panel-heading">
     <h3 class="panel-title">Create Employee</h3>
   </div>
   <div class="panel-body">

     <div class="form-group">
       <label for="fullName">Full Name</label>
       <input id="fullName" type="text" class="form-control"
              name="fullName" [(ngModel)]="fullName">
     </div>

     <div class="form-group">
       <label for="email">Email</label>
       <input id="email" type="text" class="form-control"
              name="email" [(ngModel)]="email">
     </div>

   </div>
   <div class="panel-footer">
     <button class="btn btn-primary" type="submit">Save</button>
   </div>
 </div>
</form>

{{employeeForm.value | json}}
```

component.ts

```
onSave(employeeForm: NgForm) {
    console.log(employeeForm.value);
}
```

The ngSubmit directive

... submits the form when we hit the enter key or when we click the Submit button. When the form is submitted, the *saveEmployee()* method is called and we are passing it the employeeForm. We do not have this method yet. We will create it in the component class in just a bit.

Model Driven Forms (Reactive Forms)

Form Validation

Angular disables the browser native validation on forms by default. Since the validation messages appear different in every browser, it is better to use custom validation.

Reactivate native validation

Use the ngNativeValidate directive in the opening tag of the form to re-enable browser built-in validation. The text field in the example is set to required using the HTML5 *required* attribute. If the text field is left empty and the Submit button is clicked, the form will not be submitted and the browser will display its built-in validation attribute. Else, the form will be submitted and the code in the *onSave()* method is executed, which logs the value of the text field to the console. For more information on HTML5 validation attributes, click this Link.

component.html

```
<form #textForm="ngForm" ngNativeValidate (ngSubmit)="onSave(textForm)">

        <div class="input">
            <label>Text</label>
            <input required [(ngModel)]="text" name="text" #textControl="ngModel"
type="text">
        </div>

        <div>
            <button class="inputButton" type="submit">Submit</button>
        </div>

</form>
```

component.ts

```
import { NgForm } from '@angular/forms/forms';



export class CustomersComponent implements OnInit {

  private text = "";

  ....

  onSave(textForm: NgForm) {
    console.log(textForm.value);
  }

}
```

Custom validation

Luckily, when using Two Way Data Binding, Form fields have six Properties that help with handling their State:

Pristine: Field has not been changed (it is the same as the Property's initial value)

Dirty: Field *has* been changed (it is different from the Property's initial value)

Valid: The Validation Attributes set to this Form Element have been met by the Content

Invalid: The Validation Attributes set to this Form Element have *NOT* been met by the Content

Touched: This is set to *true* when the User even so much as activates a Control (Putting the Cursor in a Textbox or Tabbing over it is enough)

Untouched: User hasn't even touched the Control

The Form itself has these six Properties, too. So it is possible to check if *any* Form Control is *invalid* or has been *touched* etc...

Using these Properties, Angular can build its own Validation Messages. This example shows the basic implementation of Custom Validation: The Textbox has the *required* Attribue, so leaving it empty will result in the *valid* Property being set to *false* and the *invalid* Property being set to *true*. The [hidden] Property of the *div* that displays the Error Message is set to the input's *valid* Property, so it is only shown when the input is *invalid*. The "Save" Button at the Bottom of the Form has the *invalid* Property of the entire

component.html

```
<form #patientForm="ngForm" (ngSubmit)="onSave(patientForm)">
    <div class="input form-group">
        <label>First Name</label>
        <input required [(ngModel)]="selectedPatient.firstName" class="form-control"
name="firstName" #firstNameControl="ngModel" type="text">
    </div>
    <div [hidden]="firstNameControl.valid"
      class="alert alert-danger inputResize">
      Please Enter the Patient's First Name
    </div>

....

    <div>
        <button [disabled]="patientForm.invalid" class="inputButton btn btn-success"
type="submit">Save</button>
    </div>

</form>
```

The following HTML can be copy pasted into a Form and used to display the values of a Form or Form Element.

component.html

Validation Properties: patientForm		
touched : {{ patientForm.touched }} **untouched : {{ patientForm.untouched }}**	**pristine : {{ patientForm.pristine }}** **dirty : {{ patientForm.dirty }}**	**valid : {{ patientForm.valid }}** **invalid : {{ patientForm.invalid }}**

Data Flow Between Components

Data Flow: Child => Parent

To send data from a child component to its parent component, Angular provides an Output mechanism with the EventEmitter class. The syntax works like this:

Declare an instance of the EventEmitter class and put the Output decorator on it. Be sure to import them both. Have the emitter emit a value in the click event of a button for example.

child.component.ts

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

...

@Output() public outputEventEmitter = new EventEmitter();

...

private onButtonClick() {
  console.log("Button Clicked");
  this.outputEventEmitter.emit('Value');
}
```

child.component.html

```
<button (click)="onButtonClick()" type="button">Click Me!</button>
```

Then, in the parent component, declare a variable to hold the value sent by the child component. Mark that variable with the input decorator (which needs to be imported aswell).

parent.component.ts

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

...

@Input() childData = "";
```

parent.component.html

```
<app-child (outputEventEmitter)="childData=$event"></app-child>
```

Data Flow: Parent => Child

To pass data into a Child Component, declare a Variable of the

parent.component.ts

```
parentData = "some string or number or complex type whatever";
```

child.component.ts

```
@Input() importedParentData = "";
```

parent.component.html

```
<app-child [importedParentData]="parentData"></app-child>
```

Data Flow between components called by Router outlet

Example on Stackblitz

Control the UI from code => Structural Directives (TBD)

ngIf *(If Statement)*

ngFor *(Foreach Statement)*

ngSwitch *(Switch Case Statement)*

Angular Specific HTML Elements

ng-content *(Allow other elements inside this element)*

ngcontent + ngif

Services and Dependency Injection

The TypeScript codebehind files in Angular are supposed to control the view *only*.To implement application logic and share data across components, Angular provides Services which can be injected into the classes with the built-in Dependency Injection.

There are a few steps to take:

Define the Service

- Use Angular CLI to create a new service
- Add the logic to the service. In this example, the service will only be used to return a static array of customer objects

customer.service.ts

```
getCustomers(){
    return [
        {"id": 1, "firstname": "Max", "lastname": "Mustermann", "company": "Muster AG"},
        {"id": 2, "firstname": "Erika", "lastname": "Mustermann", "company": "Muster AG"},
        {"id": 3, "firstname": "Hans", "lastname": "Mustermann", "company": "Arbeitslos"},
        {"id": 4, "firstname": "Tobias", "lastname": "Mustermann", "company": "Muster
Holding GmbH"}
    ]
}
```

Register the Service with the Injector

The place where the Service is registered for Dependency Injection matters, because Angular uses a hierarchical DI system. The Components are organized in a tree structure, and registering a Service in a Component (or Module) means that it will only be available in its child Components. So the best solution is to register the service is the AppModule, which is the root of every Angular App.

So Import the Service and add it to the 'providers' Array

app.module.ts

```
import { CustomerService } from './services/customer.service';


@NgModule({
  declarations: [...],
  imports: [...],
  providers: [CustomerService],
  bootstrap: [...]
})
```

Declare the Service as a Dependency in the classes where it is needed

Now add the Dependency in the Components that need the Service using Constructor Injection. Be sure to declare the Variable with a leading underscore (like *_customerService*), as this is a naming convention for variables injected with Dependency Injection. The Service can now be used to get the Customer Array. Again, this is a basic example with hardcoded data, in reality the Service would probably make an HTTP call to get some data or read the data from a database.

customers.component.ts

```
import { CustomerService } from '../services/customer.service';

...

private customers : Customer[] = [];

constructor(private _customerService: CustomerService) { }

ngOnInit() {
    this.customers = this._customerService.getCustomers();
}
```

HTTP Requests

Angular-University.io - Guide for Old Http Module, but many things are still relevant

The Same-origin policy permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin. An origin is defined as a combination of URI scheme, host name, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model. So an Angular Application making the Request by itself will throw an Error like *"No 'Access-Control-Allow-Origin' header is present on the requested resource."*. There are two ways to handle the situation: Enabling CORS, which requires Client & Server side Configuration, or using a Browser Proxy to make the Request for the Angular App, which requires Client side Configuration only.

Cross-Origin-Requests: using CORS (TBD)

CORS Client Side

CORS Server Side

Cross-Origin Requests: using Browser Proxy

The Browser can do the Request for the Angular Application, thus eliminating the need to configure the Server for Cross-Origin-Resource-Sharing. This Method only requires the Configuration of the Client. It makes use of a *.json* file to configure the Client.

- in the Application Main Folder (same Folder as .angular.cli.json), create a file for the proxy configuration, e.g. *proxy.conf.json*
- copy the following text into that file and configure the "target" parameter according to the target URL:

```
{
  "/api": {
      "target": "http://localhost:12345/",
      "secure": false
  }
}
```

This connection can now be used in a service like:

```
this.baseUrl = baseUrl ? baseUrl : '/api';      <--- access the 'api' connection

return this._http.get(baseUrl+"/data");         <--- fetch data from api endpoint "/data"
```

All requests to /api will be made to the URL specified as "target" and will

Run `ng serve --proxy-config proxy.conf.json` to launch the app and use the proxy route specified in proxy.conf.json

GET

There are several ways to parse received Objects into POCOs

Standard Method: Observables

Angulars standard method for processing HTTP Responses is turning them into Observables (or Arrays of Observables). An Observable is a sequence of items that arrive asynchronously over time (when the HTTP call returns and HTTP response).

Send HTTP Request

To send an HTTP Request, the Service needs to use the HttpClientModule. To use it, the Module needs to be added to the imports Array in add.module.ts. This makes it available to the service for Dependency Injection. It does not need to be added to the providers Array like e.g. a Service, the HttpClientModule does that by itself.

app.module.ts

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

@NgModule({
  declarations: [...],
  imports: [HttpClientModule],
  providers: [...],
  bootstrap: [...]
})
```

Inject the HttpClient into the Constructor of the Service. Depending on the Backend used, it may be desirable to configure the Service to work with JSON data as done in the Constructor here.

patient.service.ts

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

constructor(private _http: HttpClient) {
    this.headers = new Headers();
    this.headers.append('Content-Type', 'application/json');
    this.headers.append('Accept', 'application/json');
}
```

Get Observable from HTTP Response and cast it into an Array of Objects

To cast the HTTP Response into an Array of Patient Objects, the Type Patient and an IPatient Interface need to be created first. For the sake of simplicity, the Type and Interface will be declared inside the Service file.

patient.service.ts

```
@Injectable()
export class PatientService {

...

    getPatients(url : string): Observable<IPatient[]>{
        return this._http.get<IPatient[]>(url);
    }
}

export interface IPatient {
  id : number;
  firstname : string;
  lastname : string;
}

export class Patient {
  public id : number;
  public firstname : string;
  public lastname : string;
}
```

Subscribe to the Observable from the Components that need the Data

The Component that requires the Data fetched from the HTTP Server can now subscribe to the "getPatients()" Method. The Argument to the Subscribe Method is a Lambda Expression that tells the Subscribe Method to assign its data to the "customers" Array.

```
export class PatientsComponent implements OnInit {

private customers : Patient[] = [];

....

    ngOnInit() {
        this._patientService.getPatients()
            .subscribe(data => this.patients = data);
    }
}
```

"Try Parse Method"

This method allows to parse an incoming object for the desired parameters, assign the values to the POCO if the parameters are found and assign default values to the POCO if the parameters are not found. Can be used in child classes aswell.

```
export class ParentClass
{

  public name : string;
  public description : string;
  public address : number;
  public size : number;

  public array_one ? : ChildClass[];

  static fromJS(data: any): ParentClass {
        return new ParentClass(data);
  }

  constructor(data?: any) {

if (data !== undefined) {
      this.name = data["name"] !== undefined ? data["name"] : "No Name Specified";
      this.description = data["description"] !== undefined ? data["description"] : "No
Description";
      this.address = data["address"] !== undefined ? data["address"] : 1;
      this.size = data["size"] !== undefined ? data["size"] : 10;

      var parse_array = data["array_one"] !== undefined ? data["array_one"] : null;

      this.array_one = new Array<ChildClass>();
      for (let index = 0; index < parse_array.length; index++) {
        const element = parse_array[index];
        let newelement = new ChildClass(element);
```

```
          this.array_one.push(newelement);
        }
      }
    }
  }

  export class ChildClass
  {

    public childname : string;
    public childaddress : number;

    static fromJS(data: any): ParentClass {
          return new ParentClass(data);
    }

    constructor(data?: any) {

    if (data !== undefined) {
        this.childname = data["childname"] !== undefined ? data["childname"] : "No Child Name
  Specified";
        this.childaddress = data["childaddress"] !== undefined ? data["childaddress"] : 0;
    }
  }
```

The expression used here is evaluated as follows:

```
        this.name = data["name"] !== undefined ? data["name"] : "No Name Specified";
```

- if the json object has a property called "name" then assign its value to this.name

- if the json object does not have a property called "name" then assign "No Name Specified" to this.name

POST

When working with an ASP.NET Web API & Entity Framework (like my PatientManagement Project), the Primary Key (in my case "patientId" must be set to 0 for a POST Request to work. The Api Controller will check the Value of the Primary Key and return a 400 Bad Request if the key is NULL!

Preflight the HTTP POST Request (TBD)

Subscribe to the POST Observable

To save a newly created Object to the Backend, it is not enough to simply send a Patient Object to the PatientService and have the Service make a POST call. The HTTPClient in Angular will only execute that REST Call, if the Angular Application subscribes to the Observable that is returned by the HttpClient.post Method. To execute the call properly: have the Method in the Service that calls the HttpClient.post Method return the result of the HTTP call: `return this._http.post(this.baseUrl + "patients", body, this.httpOptions);`. Then, subscribe to the response (which is an Observable) in the Component Method: `this._patientService.postPatient(this.selectedPatient).subscribe(...);`. It is not necessary to assign the Response to a Variable, the *subscribe* Method can just make a console log for example.

patient.component.ts

```
onSavePatient() {
    this._patientService.postPatient(this.selectedPatient)
                        .subscribe(() => console.log('patient saved successfully'),
                        console.error
                    );
  }
```

patient.service.ts

```
postCustomer(patient: Patient) {
    let body = JSON.stringify(patient);
    return this._http.post(this.baseUrl + "patients", body, this.httpOptions);
}
```

Routing

To navigate between views, Angular provides a Router. To use it, several steps are necessary.

Please note: This example was copied from a YouTube Tutorial. To copy & paste these examples, your Angular app requires a *customers* component and a *products* component. Otherwise, the code snippets need to be changed according to your actual application.

In app.module.ts, import the RouterModule and Routes Object from @angular/router. Also declare an Array of type Routes that will hold the routes for the application. Now add the routes as shown below in the example for two components "customers" and "products". The last item in the example Array is used to configure what happens if the base URL of the application is called. In this case, the app will reroute the user to the customers route. Then add the Routermodule to the imports Array, passing the appRoutes Object using the forRoot method.

app.module.ts

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


const appRoutes: Routes = [
  { path: 'customers', component: CustomersComponent },
  { path: 'products', component: ProductsComponent },
  { path: '', redirectTo: '/customers', pathMatch: 'full' }
];

@NgModule({
  declarations: [...],
  imports: [
    ...
    RouterModule.forRoot(appRoutes),
    ...
  ],
  providers: [...],
  bootstrap: [...]
})
```

Routing from a Button Click

In the app.component.html, create a nav bar that holds some buttons which will call the routes. Configure the routes that shall be called using the `routerLink` directive provided the by the RouterModule. The `<router-outlet></router-outlet>` component specifies where the components that were specified in the `appRoutes` Array will be shown.

app.component.html

```
<nav class="navbar navbar-default">
    <ul class="nav navbar-nav">
        <li><a routerLink="customers">Customers</a></li>
        <li><a routerLink="products">Products</a></li>
    </ul>
</nav>
<router-outlet></router-outlet>
```