

## LABS AND EXERCISES ANGULAR FUNDAMENTALS

### 01 – START SIMPLE

The general Github-address with example code is [github.com/PeterKassenaar/voorbeeldenAngular2](https://github.com/PeterKassenaar/voorbeeldenAngular2)

- a) **Goal: create a basic Angular Hello world app.**
  - Create your own Hello World app. This can be done in multiple ways. Choose one:
  - Use `100-helloworld` as a base. You only need to install dependencies by using `npm install` and subsequently `npm start`.
  - OR (recommended!): Install Angular CLI and create a project using the command line.
  - Directions can be found at <https://cli.angular.io/>.
  - Learn to use the commands `ng new`, `ng serve` and `ng generate`. Look up for yourself on how to use these commands.
- b) **Goal: create a new component. Show this inside the root component.**
  - Make sure Angular cli is installed. Use `npm install -g @angular/cli` if this is not the case.
  - Use the angular CLI to create a new component. For instance: `ng generate component [your-component-name]`.
  - Open your component in the editor and give it some UI (add some HTML text).
  - Look at the `.ts`-file. Remember its selector.
  - Go to the main component of your app and enclose the new component using its selector (like `<your-component></your-component>`).
  - Start the application. It should show both components.
- c) **Optional Goal: write your own component. Use this instead of the default `app.component.ts`.**
  - Remove the `<hello-world>` or `<app-root>` component from your existing app. Then, follow these steps to create your own, new component from scratch:
  - Using Angular-CLI (recommended): again use the command `ng generate component [cmp-name]` and edit some text inside the generated `.ts` and `.html`-files.
  - Edit the Module `app.module.ts`, such that the new component is imported and bootstrapped.
  - Edit `index.html`, such that the new selector is being used.
  - Run `npm start` to test your new component.
- d) **Optional Goal: getting to know TypeScript a bit better**
  - Visit [typescriptrlang.org](https://typescriptrlang.org) and practice creating a class, using the Playground link in the Top Menu. Type your code in the left pane, look at the generated code in the right pane.
  - Define for example a class `Person`, with properties `firstName`, `lastName` en `email`.
  - Create an instance of the class (`let person = new Person()`) and log the properties to the console. Use the **Run** button above the right pane.
  - See for yourself for the other examples from the dropdown menu. (like `Types`, `Generics`, `Modules`) to extend your knowledge of TypeScript.
  - Visit the link `Documentation, Handbook, Basic Types` and read what types can be used in TypeScript.
  - If time permitting, also look at the other documentation (`Variable Declarations`, `Interfaces`, `Classes`, etc).

## 02 – DATABINDING

**a) Goal: create properties on your class and bind them to the user interface.**

- Expand your app from the previous lab (Hello World) with a field/property. Bind the property in the template being used.
- *First*, use direct initialization of variables, like `name: string = '<your-name>'`.
- *Second*, use separate declaration and initialize in the constructor. Code can look like:  

```
name: string;  
constructor() { this.name = '<your-name>' }
```
- *Third*, use the (recommended) approach using `ngOnInit()`. Code can look like  

```
ngOnInit() { this.name = '<your-name>' }
```
- Demo code available at `/101-databinding`.

**b) Goal: working with arrays of properties and using `*ngFor`.**

- Create an array of properties. Bind them in the template using the directive `*ngFor`.
- Use TypeScript to explicitly declare the property as an array of strings. Pick one of these alternatives (they both mean the same):  

```
cities: string[];  
cities: Array<string>;
```

Of course you can use other data than cities, for example `persons`, `products`, and so on.
- Demo code available at `/101-databinding`.

**c) Goal: create your own Model and use this as a type in your application.**

- Create a `Model` for the contents of your array. The model consists of an object with one or more properties. See for instance `app/shared/city.model.ts` as an example.
- Adapt the signature of the array so it looks like `cities: City[]` or `cities: Array<City>`.
- Rewrite your array, so the content now are properties of type `<YourModel>`.
- Advanced: instead of using a class for the `Model` you can also use the TypeScript-construct `Interface`. Look up for yourself how this can be done.
- Demo code available at `/101-databinding`.

**d) Goal: using `*ngIf`**

- Create a `<div>` on the page that is only shown if your array has three or more objects in it. The code can look like `<div *ngIf="cities.length > 3">...</div>`.
- Create one more use case of `*ngIf` for yourself.

**e) Goal: using an external template**

- Edit your component, such that the HTML is now in an external template. In the example code this is in `app/app.html`.
- Next, use the property `templateUrl` in `@Component` to point at the file. Test if the databinding still works.
- Demo code available at `/101-databinding`.

## 03 – EVENT BINDING

### a) Goal: using event binding

- Add an element with event-binding to the application. For instance, create a button to capture a click event. The code can look like  
`<button (click)="onClick()">...</button>`.
- Call an event handler in the component if the event occurs. For example, write a function `onClick() { alert('message...') }` or show a `console.log()`-text.
- Demo code available at `/103-eventbinding`.

### b) Optional goal: using other event bindings

- Test other types of DOM-events, for example `blur`, `focus`, `keypress`, `mousemove`, and more. If you need inspiration, see the overview of possible events at [mozilla.org](https://developer.mozilla.org/en-US/docs/Web/Events) (<https://developer.mozilla.org/en-US/docs/Web/Events>).

### c) Optional goal: using `@HostListener`

- Use the `@HostListener()` decorator in your component to listen to other (non-DOM) events, like networking changes, battery charging change or resize events.
- Show a message in the console or on the component if the event occurs.

### d) Goal: learn local template variables

- Create a text field with a local template variable. The notation can look like  
`<input type="text" #myText>`.
- Pass the variable to an event handler using an event of your liking (for example `click` or `keyup`) and show the value in an `alert()` or `console.log()`
- Create another textbox on the page. The user can type numbers in the textbox.
- Pass the number to an event handler and add the number to a property `total`. Show the addition of all numbers (i.e. the total value) in the page.
- Remember to use `parseInt()` to convert the stringvalue of the textbox to a number!
- Demo code available at `/103-eventbinding`.

### e) Goal : creating a simple client sided CRUD application

- Create a simple client-sided CRUD-application: users can add elements to the array (names, cities, products, etc) and remove them from the array.
- Adding can be done by using `ArrayName.push(...)`
- Deleting can be done by using `.indexOf()` and `ArrayName.splice(...)`, OR: using a more reactive approach, by using `.filter()` operator on the array.

## 04 - ATTRIBUTE BINDING AND TWO-WAY BINDING

### a) Goal: getting to know attribute binding

- Create a button on the page. If the button is clicked, a `<div>` with a text is shown.
- If the button is clicked again, the text is hidden.
- Do this by adding a property to the class and assign it to the `[hidden]` property of the div.  
For example:  

```
<div [hidden]="myBoolean">...</div>.
```
- Demo code available at `/103-attributebinding`

### b) Optional goal: using other attribute bindings

- If time permitting: create a component with a textbox.
- If the user types an English color in the box and clicks a button, a corresponding `<div>` receives this color as a background color.
- Hint: use `[style.backgroundColor]="bgCcolor"` on the div. Create a `bgCcolor` property on the class.
- Optional: create a second textbox to set the text/foreground color.
- Advanced: investigate how this works if the color can be picked from a series of radio buttons or from a dropdown list.

### c) Optional goal: using class binding and style binding

- If time permitting: investigate for yourself how the Angular-concepts *class binding* en *style binding* work. Read for example the documentation at <https://angular.io/docs/ts/latest/guide/template-syntax.html#!#other-bindings>.
- Create a component that demonstrates these concepts. Show and explain your code to the teacher or to a colleague.

### d) Goal: using two-way binding with `[(ngModel)]`

- Create a text field in your component that uses two-way binding.
- Use `[(ngModel)]` as a directive on the `<input>` box. Bind the value of the typed text directly to the page.
- Don't forget to import and add `FormsModule` to the `app.module.ts` file!
- Create a copy-function. Create two text boxes on the page. Text that is typed into the first textbox, should also appear in the second textbox.
- Demo code available at `/104-twowaybinding`

## 05 - SERVICES

### a) Goal: using a service to display data in component/user interface.

- While working, you have developed some components, displaying data. Move this data from the component class to a service.
- Create a new file (for example `city.service.ts`) and import the correct annotations. Use `@Injectable()` to decorate the service.
- Write a `getCities()` method and optional methods to add or delete cities - or the data you are working with, of course.
- Inject your service in the module. Remember to use the property `providers: []` in the `@NgModule`-annotation.
  - i. Optional: remove the providers array and update the decorator on the service to look like `@Injectable({providedIn: 'root'})`.
- Import your service in the component and instantiate in the constructor. The code will look like `constructor(private cityService: CityService) { ... }`. Test your application.
- Demo code available at `/200-services-static`.

### b) Optional goal: provide other data via service.

- Create another service, providing different data (for example your favorite food, sports teams, hobbies, etc).
- Follow the same steps as above, consuming the data from the service in your component.
- This is an illustration of the fact that you can inject multiple services into the constructor of the same component.

### c) Goal: use a service to fetch contents of a .json file asynchronous.

- Create a .json file with data and load this data in your application using an asynchronous service call (for example: `cities.json`). Remember the following steps:
- Import and inject `HttpClientModule` in `app.module.ts`.
- Inject `HttpClient` in the service;
- Adapt your get-method. For example, `getCities()` now has this body:
 

```
return this.http.get<City[]>('app/cities.json');
```
- Adapt your component, to use a `.subscribe()` method to retrieve and unwrap the values from the service. Our subscribe block can look like
 

```
.subscribe(cityData => this.cities = cityData)
```
- Demo code available at `/201-services-http`.

### d) Optional goal: learn to work with other RxJs methods.

- More on RxJs: use the `.pipe()` function and use other operators inside `.pipe()`. For instance:
  - i. `map()`, `delay()`, `tap()`, `catchError()` among others.
- Demo code available at `/202-services-rxjs`.
- If time permitting: read the Observable Cheat Sheet at [onehungrymind.com/observable-cheat-sheet/](https://onehungrymind.com/observable-cheat-sheet/). See what observables can do for you and how you can transform the data stream to be directly consumed by your components.

### e) Goal : using a live API

- See how a service connects to a real backend by running and studying `/210-services-live`. Look at:
  - ...how the service connects to the backend by using the URL;
  - ...how the results returned from the Component or Service.

- Having studied the example, build an app on your own! You can of course use the Open Movie Database API, but there are a lot of other API's available.
  - See for example [openweathermap.org/API](https://openweathermap.org/API), or <https://pokeapi.co>
  - See also the textfile `JavaScript APIs.txt` and even more API's at <https://github.com/toddmotto/public-apis>.
  - Remember to use `HttpClient` in your service
  - Always read the API documentation **very carefully!**
- f) **Optional goal : sending POST requests to a backend**
- See how a service sends POST requests to a real backend by running and studying `/212-post-demo`. Look at:
  - ...how the component now directly connects to the backend by using the URL;
  - ...how the data attribute is built from the dummy `email` and `password` fields;
  - ...how the `headers` field is composed and added to the POST request.
  - Having seen this, go ahead and create your own POST-demo, using <http://regres.in>.
  - There are POST-endpoints available for dummy creating records, dummy registration and dummy login successful/unsuccessful. Pick one and assemble a component talking to this backend.
  - Demo code available at `/212-post-demo`.
- g) **Optional goal: creating a backend with json-server**
- You can use the package `json-server` to create a 'fake' REST backend. Well, actually it is a real backend, but a `.json`-file serves as your database. You wouldn't want this in a real application, but it is useful in development scenarios.
  - Install `json-server` in you application, using `npm install json-server`.
  - Read the documentation at <https://www.npmjs.com/package/json-server>
  - Create your own `.json` database-file and serve it from `localhost://3000`
    - i. You can optionally use the file `MOCK_DATA.json` as an example, if you can't come up with your own data.
  - Run your frontend application from `localhost:4200`
  - Show a list with the contents of the `json`-file in your component.
  - Create a 'real' CRUD-like application, where your application can read, write and update contents inside the `json`-file.
  - Demo code available. See `./205-services-http-CRUD` as an example.

## 06 - APPLICATIONS WITH MULTIPLE COMPONENTS

### a) Goal: creating a detail component showing the selected data.

- Create a new application, or use your application from previous exercises.
- Add a detail component. Details are shown after a mouseclick on the list in the parent component.
- Follow the steps (4 steps) from the presentation, to add the DetailComponent to your app. These are:
  1. Create a separate detail component, for instance `city.detail.ts`.
  2. Inject detail component in `app.module.ts`.
  3. Encapsulate detail component in HTML of parent component by using its selector, for instance `<city-detail></city-detail>`. Also alter component so that detail component is loaded after mouseclick on the list.
  4. Run the app and look if detail component appears after clicking
- Demo code available at `/300-components`.

### b) Goal: using `@Input()` annotation to transfer data from parent to child component.

- Add/enhance the click handler in your parent component: make sure the current city (or again: the data you are working with) is transferred to the component (i.e. `(click)=getCity(city)`).
- In your event handler, set a property. For instance `this.currentCity = city`.
- Make sure to transfer the `currentCity` property to the child component. Use a notation like `[city]="currentCity"` in the template of the parent component.
- Annotate the detail component with `@Input()` decorator. Don't forget the `()` after `@Input`. This way, the `City` object is passed into the detail component and shown in the user interface.
- Demo code available at `/301-components-inputs`.

### c) Goal: using `@Output` annotation to transfer data from child to parent component.

- Import `@Output()` service and `EventEmitter` in the child component using `import {..., Output, EventEmitter} from '@angular/core';`
- Enhance the class with `@Output()` decorator and define a new `EventEmitter` (see `city.detail.ts` in example code).
- Write a click handler that emits a new event. In the example code a `number` is thrown, but you can throw anything you like.
- Capture the event in the parent component by using event binding syntax like `(eventName)="handleEvent($event)`.
- Write an event handler in the parent component to handle the event and let the application react accordingly.
- Demo code available at `/302-components-outputs`.

### d) Optional goal: create a complete application:

- If time permitting: create a simple eCommerce application, implementing the following requirements:
  - A Store. Write a service for a store that delivers 4 or 5 products. (This can be static data).
  - A master component. Show a list of items in the store, retrieved by your service.
  - A detail component. Clicking a product shows details of the product in its own component.
  - A Shopping Cart. The user can place the product in his shopping cart (which of course is another component). The contents of the shopping cart are visible in user interface.
  - An "Order" button. If clicked, show the contents of the shopping cart and calculate the total price.

## 07 - ROUTING

### a) Goal: install routing in your app.

- Adapt the application from your previous exercises, or create a new application, using routing. These are the requirements:
- Add the `<base href="/">` tag to your `index.html`.
- Create a routing table in a new file called `app.routes.ts`. Make sure to export a `constant` that is an array with routing properties.
- Import your routes in `app.module.ts`. Don't forget to also import `RouterModule`.
- Make sure the application has a `mainComponent` (also known as: `rootComponent`). This component shows the main menu and has a `<router-outlet></router-outlet>`.
- Adapt the bootstrapper `main.ts`, such that now the new `mainComponent` is loaded/bootstrapped.
- Adapt `index.html`, such that the correct selector is loaded.
- Create the (new) component for every route you designed. This can be `EditComponents`, `addComponent`, `AboutUsComponent`, `ContactComponent`, and so on.
- Test and make sure the different components are loaded when they are selected in the main menu.
- Demo code available at `/400-router`.

### b) Optional Goal – using routing with Angular-CLI

- Create a new project with the Angular-CLI, and use the `--routing` flag to add routing to the project.
  - i. OR: when Angular CLI v.7, answer (Y)es if prompted if you want to add routing to a new project with `ng new <project-name>`.
- Study the separate routing module that the CLI created and see how it is integrated in the app.
- Add one or two new components (and routes) to the application and show them via a main menu inside `<router-outlet>` in `AppComponent.ts`.

### c) Goal: use routing parameters

- Adapt your `app.routes.ts`, such that now a route with parameter is available. This can be written like

```
{path: 'detail/:id', component: CityDetailComponent}
```
- Add a new component, for instance `city-detail.component.ts`. Make sure to import `ActivatedRoute` into this component.
- Instantiate a private route in the constructor, of type `ActivatedRoute`. The code can look like

```
constructor(private route: ActivatedRoute) { ... }
```

  - i. Make sure your detail component is added to the Module, so it is available.
- Adapt the master page, to point to the detail route. Hyperlinks can look like

```
<a [routerLink]="['/detail', city.id]"> {{ city.name }}</a>
```
- Save and test your app.
- Demo code available at `../401-router-parameter`.

### d) Optional goal: securing parts of your application with RouteGuards.

- If time permitting: add some guards to your application. For instance, the `CanActivate` guard or `CanDeactivate` guard.
- Write a *guard as a function*, using the `provide:` and `useValue:` properties of the `providers: []` section in `app.module.ts`.
- Remember to use/add the guard token in `app.routes.ts`.



- Write a guard as a class, using the `CanActivate` interface. See for instance `canActivateViaAuthGuard.ts` as an example.
- Again, don't forget to add the guard in `app.routes.ts`.
- Try to let the example `authService` return `true` or `false` and see what happens. Can you still navigate to the route?
- Demo code available at `../402-router-guards`.

## 08 - FORMS

### a) Goal: build a template driven form

- Create a simple HTML5 form in a component, or use a form from an earlier exercise (for example the login form from 11-post-demo).
- Add the local template variable `#myForm="ngForm"` to the `<form>` tag.
- Add the directives `ngModel` to the separate form fields. You don't need two-way databinding with `[( )]`.
- Write for example `myForm.value` to the user interface, or show the contents of the form in an alert (or in the console) when a button is clicked.
- Demo code available at `/500-forms-template-driven, Component 1`.

### b) Goal: address individual controls inside the form and add HTML5 validators.

- Assign a local template variable to the form fields.
- Bind `ngModel` to the local template variable. The code can look like `#email="ngModel"`
- Retrieve the values from the local template variable and show them in the user interface, for example its value and its validity.
- Add the HTML5 attribute `required` to the form fields and see how this affects the state of the form field. Write its validity to the user interface.
- Demo code available at `/500-forms-template-driven, Component 2`.

### c) Goal: combining individual form fields to an `ngModelGroup`

- Add some field to the form (for example some extra text fields or checkboxes).
- Groep them inside a `<div>`, assign the `<div>` the directive `ngModelGroup`. The code can look like   
`<div ngModelGroup="customer" #customer="ngModelGroup">`
- Run the code and identify the model group in the returned form value object.
- Optional: set the value of a form field from inside your class, by using the local template variable and bind to `[ngModel]`.
- Demo code available at `/500-forms-template-driven, Component 3`.

### d) Goal: submitting template driven forms

- Add a submit button to the form.
- Make sure the submit button is only active when the form as a whole is valid. Your code can look like   
`<button type="submit" (click)="onSubmit(myForm)" [disabled]="!myForm.valid"> ... </button>`
- Demo code available at `/500-forms-template-driven, Component 4`.

### e) Optional goal: working with model driven forms

- Start with a simple form, for example build a form on your own, or use the dummy login form from `/212-post-demo`.
- Import `ReactiveFormsModule` into your `app.module.ts`.
- Add the `[formGroup]="..."` directive to the `<form>` tag, add `formControlName="..."` to the individual controls.
- Import `FormGroup` and `FormBuilder` into your class and build the form, based on the layout of your HTML.
- Submit the form and write the value to an alert box or to the console.
- Demo code available at `/501-forms-model-driven, Component 1 and Component 2`.