Assignment 8 – Angular Guide

Vladimir A. Shekhovtsov (volodymyr.shekhovtsov@aau.at)

Contents

TypeScript and ES6

The lectures should be enough to complete the exercise 1, below is some additional information which is important for Angular development.

Modules

You have probably already dealt with modules while working with Node.js and Express, here is a short recap, as they are important for Angular development as well. Modules are not a TypeScript feature; they are also present in ES6.

Modules are defined on the file level, i.e. the `.js` or `.ts` file constitutes a module when it is written in a specific way which I describe below. Every module defines its own scope; this means that variables, functions, classes, etc. declared in a module are visible outside the module only if we **export** them explicitly. On the other hand, to access a variable, function, class, interface, etc. exported from a different module, it is necessary to **import** it explicitly.

The rule for defining the module is as follows: *a module is a* `.js` *or* `.ts` *file containing a top-level import or export declaration*. Note: if no such statement is present, the file is considered a script, and everything within it is in a global scope.

To export anything from a module, it is necessary to precede its declaration with an export keyword to make an export declaration:

```
// x.ts: a module
export class X {
  // X is accessible outside this file to the modules which import x.ts
  constructor() {}
}
export globalVar: number = 0;  // exported variable
```

To import anything from a module, it is necessary to provide an import declaration. For the specific export, it looks like this:

```
// y.ts: another module as it starts with import
import { X, globalVar } from "./x"; // no file extension has to be specified
//...
let x1: X = new X();  // OK, X is available
let x2 = globalVar;   // also OK
```

It is also possible to input everything which is exported:

```
import * from "./x";
```

if the module contains a lot of exports, importing everything with their names can be complex to manage (a lot of new names will be introduced). It is recommended to *import module as a variable*, and then use the name of the variable to access the module exports:

```
import * as XOperations from "./x";
let x = new XOperations.X(); // OK
```

Decorators

Decorators allow to specify a meta-information about the class or its member which can be used by some external processor, not by TypeScript compiler or JavaScript interpreter. We will only learn how to use the existing decorators, as Angular depends heavily on that, these decorators contain information important for Angular environment.

The application of the decorators always begins with a `@` following with the name of the decorator. Some decorators do not accept parameters, to apply such decorator to a class you need to precede a class name with `@decoratorName`:

```
// standard decorator for injectable objects in angular, see exercise 4
@Injectable
export class X {  … }
// now X has additional features defined by the creator of the decorator
```

Other decorators accept additional parameters, it is necessary to specify such parameters in parentheses following the name of the decorator: `@decoratorName(parameters)`:

```
// Angular module decorator which specifies
// the additional properties of the class which it precedes, see exercise 2
@NgModule({   // a JSON-defined object is a parameter of the decorator
  declarations: [AppComponent],
  imports: [],
  providers: [],
  bootstrap: [AppComponent]
})   // the end of the decorator
export class AppModule { }  // the class itself
```

Here you see an example of applying the decorator with parameters, the parameter here is just an object defined by JSON notation, the class definition is in the last line, the class is actually empty as everything which is necessary is defined within the decorator parameter.

Member-defining constructor parameters

To save effort for defining members which must be initialized in constructors, it is possible to define such members right in the parameter lists of constructors by preceding the parameter specification with the access modifier keyword (private/public/protected):

```
class X {
    constructor (private xMember: number) {}  // member-defining parameter
}
```

This is actually the same as:

- declaring the member in a class body
- passing the parameter to a constructor
- initializing this member with this parameter value

Dealing with interfaces: implementing attributes with setters and getters.

Interfaces define attributes and member functions without specifying access modifiers (private/public/protected). If the member attribute is specified, it has to be implemented, usually with a private attribute and a public setter-getter pair:

```
interface IX {
    xAttribute: number;
}
class X implements IX {
    private _xAttribute: number;
    public get xAttribute(): number { return this._xAttribute; }
    public set xAttribute(newxAttribute: number) {
       this._xAttribute = newXAttribute;
    }
}
```

Optional members

By default, interface members are mandatory to implement. It means that if the member is defined in the interface, it must be implemented in all the classes which implement this interface, otherwise the compiler will return an error. It is also possible to specify the optional members, the implementation of such members can be omitted from the implementing classes. Such members are defined by appending a question mark to their names (before the semicolon, or before the parentheses for functions):

```
interface IX {
    public xOptional?: number;
    public xOptionalFunction?(): number;
}
```

Classes can also have optional members; it means that such members can be omitted when the instance of the class is created e.g. by means of the JSON notation:

```
class X {
    public xOptional?: number;
    public xMandatory: number;
}
let x1: X = { xOptional: 10, xMandatory: 100 } // OK
let x2: X = { xMandatory: 100 } // OK, omitting optional members is possible
let x3: X = { xOptional: 10 } // not OK, missing mandatory member
let x4: X = { } // not OK
```

Parameterized types

TypeScript allows to define parameterized (template-based) types similar to those in C++ and Java. We will not need to create our own parameterized types for this assignment, but we will parameterize the existing ones coming from Angular framework. The syntax is as follows: `TypeName<TypeParameter>`, note that the syntax for defining arrays also follows this convention, so they are actually also the parameterized types in TypeScript:

```
let a: Array<number> = [];
let b: OtherLibraryType<string> = new OtherLibraryType();  // something else
```

Note that the constructor calls need not be parameterized.

Backtick strings

A convenient way of dealing with strings in TypeScript is to use the string literals which are enclosed in backticks: `` `string` ``. Such strings have the following advantages:

1. they can span lines:

```
let s =
`this is my text
 and more of that
 and still more`;
```

2. they allow for expression substitution, i.e. it is possible to embed the JavaScript expression within such a string and make the value of such expression substituted as a part of the string at runtime. For simple variables it is enough to precede the name of such variable with a dollar sign: `$variable-name`:

```
let x = 100;
console.log(`the value is $x`);  // outputs the value is 100
```

   More complex expressions should be enclosed in the curly brackets following the dollar sign: `${expression}`

```
let x = 100; let y = 200;
console.log(`the value is ${x+y}`);  // outputs the value is 300
let z = { field: "xxx" };
console.log(`the value is ${z.field}`);  // outputs the value is xxx
```

Exercise 2 – Getting started with Angular

Generating a new application

Generating the application with the Angular CLI by means of `ng new project-name` creates the special predefined directory structure. Let us look at this structure in more detail.

A. The project root is a `project-name` directory created below the directory where `ng new` was run. It contains files which are necessary for building and managing the project such as `package.json`, `tsconfig.json`, and `angular.json`. Most of the time, these files are managed automatically by the Angular CLI.

B. There are several directories below the root, we for now will only deal with the `src` directory which contains the source code of the application (the others are e.g. `node_modules` directory containing the Node.js dependencies for the project or `e2e` directory containing the files related to application testing).

C. The `src` directory directly contains the files which are necessary for starting the application, but which are rarely touched by the developer. Among them are:

   a. `main.ts` – the startup script of the application, it includes the bootstrapping call which launches the Angular environment. While doing this, it loads the application module described below. It is almost never necessary to change anything in this file.
   b. `index.html` – the root template of the application which is used by the browser to render the top-level display. It contains the custom tag (usually `<app-root></app-root>`) which renders the application component, again described below. If you want e.g. to change the title tag of the application, it can be done there.
   c. `styles.css` – the root style file of the application, defines the styles available anywhere.

D. There are three directories further below the `src` directory:

- the `app` directory contains the code of the application
- the `assets` directory contains the external resources necessary for the application
- the `environments` directory contains the files defining different development environments (e.g. one for development and one for production).

For now, we will only deal with the `app` directory (further on, we will assume that all our paths start from `app`, so it will not be included in these paths, we will use e.g. `file.ts` instead of `app/file.ts`). In a new project, it directly contains the following four files: `app.module.ts`, `app.component.html`, `app.component.ts` and `app.component.css`. The first file defines the **application module**, the other three – the **application component**.

The application module

The Angular modules define the dependencies shared by all the code belonging to the module and the capabilities provided by the module to other modules or to the Angular framework (such as the set of implemented components). The application can contain more than one module, but a single module is enough in many cases, we will not deal with multi-module configuration in the assignment.

The application module defines the set of root dependencies in the application and the capabilities which are provided to the Angular framework. Every module is defined as an (usually) empty class with the `@NgModule` decorator applied to it; all module configuration is specified by the parameters of the decorator. This parameter is split into several sections described by the comments in the below code.

```
// app.module.ts
@NgModule({  // the Angular module decorator
  declarations: [  // the components which are provided by this module
    AppComponent   // the application component, see below
  ],
  imports: [   // the modules which are necessary for this module
    BrowserModule   // a standard module which is always included
  ],
  providers: [],  // the services which are provided by the module,
                  // see exercise 4
  bootstrap: [AppComponent]  // the component which is executed at startup
})
export class AppModule { }  // the module itself defined as an empty class
```

The application component and the component files

Three files: `app.component.html`, `app.component.ts` and `app.component.css` are related to the *application component* for the project. Every component is responsible for rendering some fragment of the application page (*the component view*), the application component is unique as it is responsible for rendering the “root” view of the application i.e. the view which is available directly from its root page `index.html` and usually occupies the whole browser window. It is possible to refer to other components from this view (see exercise 3), so their views are nested within it, this can be repeated recursively, i.e. these other components may refer to other components forming the component hierarchy, which is, as a rule, reflected visually by nesting their views.

The responsibilities for these three files are the same for every component:

- *`name`*`.component.html` defines **a component template**. It looks like the html page with some extra tags, tag properties, and placeholders, all these extra tags and placeholders are pre-processed with Angular to form the final fragment of the html page which is substituted in place of the component selector tag to eventually form the final html page which is rendered by the browser.
- *`name`*`.component.ts` contains **a component class** which defines the dynamic content shown in a template and the user interaction with this template, the values of its properties are substituted for the placeholders, its methods are called to handle events happening within its view (e.g. the user clicking on a button etc), other methods are executed at some moments while the page is rendered (e.g. on page load).
- *`name`*`.component.css` defines **a component style file** which is applied to the processed template, so it is possible for the resulting html to use styles defined in this file.

The exact names of the template and style files, and the name of the component tag are defined within the parameter of the `@Component` decorator applied to the component class:

```
// app.component.ts
@Component({  // the component decorator
  // this is the selector tag which will be used to render the component
  // as <app-root></app-root>, see index.html
  selector: 'app-root',
  // this is the template file for the component
  templateUrl: './app.component.html',
  // this is the set of style files for the component
  // (more than one style file is possible)
  styleUrls: ['./app.component.css']
})
export class AppComponent { ... } // this is the component class itself
```

For every component class, its instance is implicitly created by Angular when the page with component's tag (its selector) is rendered. As a result, the initialization of the class variables and call to its constructor are performed on each reload of the corresponding page.

Angular also provides the lifecycle hooks which are automatically called on different stages of the page lifecycle, e.g. if we redefine the `ngOnInit()` method for the class (you can see in the generated component classes), it will also be called on opening the page.

Displaying the information in the component view

To output anything coming from the component class to the component view it is necessary:

1. to change the template to include the placeholders for the dynamic data coming from a component class
2. to change the component class defining the information substituted within the template.

The simplest kind of placeholders which can be included in a template are the placeholders which refer to the properties of the component class directly by name. The placeholders in Angular templates are enclosed in double curvy braces: `{{}}`. A placeholder referring to a single property looks as follows: `{{ property-`*`name`*` }}`. Suppose we have the following template fragment

```
<div>template text<strong>{{ myProperty }}</strong></div>
```

Now if our component class contains the property defined as follows:

```
@Component({...})
export class MyComponent {
   myProperty: string = "my text";
}
```

the html source of the rendered page fragment will look as follows

```
<div>template text<strong>my text</strong></div>
```

Any changes in the class property are immediately reflected visually, so if we assign anything to the property: `this.myProperty = "other text"`, the browser display will *immediately* reflect this change.

Note that spaces inside placeholders are ignored while rendering as the placeholders contain TypeScript expressions and not html markup.

Defining the data

The angular data definition convenience is to put all classes defining the data into the separate directory under `app`, usually it is called `model`. Here, the data is usually represented as plain TypeScript classes defined within modules:

```
model/thing.ts
export class Thing {
   name: string;
   good?: boolean;   // optional
}
```

Then these classes are imported into necessary modules:

```
// the path should reflect the directory structure
// e.g. assume that we are in the module which is within the directory
// under app, then to refer to the model/thing.ts we need
import { Thing } from "../model/thing";
```

Good IDEs like IDEA could handle this automatically by finding modules within a project after encountering their exports in the entered code and inserting import statements with correct paths. I.e. you can write `let d: Thing`, see `Thing` underlined by the IDE (which means that the export with this name is available in a project), press a key (Alt-Enter in IDEA) and make the import statement written for you.

Now it is possible to create instances of such classes within component code

```
export class MyComponent {
    public thingList: Thing[] = []; // empty array
    constructor() {
      this.thingList = [
        { name: "the first thing", good: true },
        { name: "the second thing", good: false }
      ];
    }
}
```

Now it is possible to refer to the values in the template, any JavaScript syntax is allowed within substitution brackets `{{ expression }}`

```
<div>see {{ thingList[0].name }}</div>
```

will render the component display as follows

```
<div>see the first thing</div>
```

Loops and conditionals in templates

Often it is necessary to repeat some tag for the elements of an array e.g. for rendering the list of elements using the template fragment representing just one element. This is supported by using the directive `*ngFor`. This directive looks like an html attribute, but its value specifies a loop statement similar to a for loop: `*ngFor="let` *`variable`* `of` *`array`*`"`. As a result of processing such directive, Angular substitutes the tag containing `*ngFor` as many times as there are elements in the *`array`* and makes its current element available as *`variable`* within the tag.

The following template fragment repeats the `<div>` element containing `*ngFor` directive for every element in `Thing` and makes its current element accessible as `x` within the `<div>`

```
<div *ngFor="let x of thingList">
   see {{ x.name }}
</div>
```

If `Thing` contains the data as described in the previous subsection, the above template fragment will be represented in the resulted html source as follows:

```
<div>
   see the first thing
</div>
<div>
   see the second thing
</div>
```

It is also often necessary to show or hide the element depending on a condition. This is supported by using the directive `*ngIf`. It again looks like an html attribute, but this time its value contains a conditional TypeScript expression (returning true/false): `*ngIf="`*`condition`*`"`. As a result of processing this directive, Angular includes the tag containing `*ngIf` in the resulting markup only if the *`condition`* is met.

This fragment will add the `<span>` to the resulting markup only if the elements are good:

```
<div *ngFor="let x of thingList">
 see {{x.name}}<span *ngIf="x.good"> (good)</span>
</div>
```

If `Thing` again contains the data as described in the previous subsection, the above template fragment will be represented in the resulted html source as follows:

```
<div>
   see the first thing<span> (good)</span>
</div>
<div>
   see the second thing
</div>
```

Note that `<span>` elements are often used to show/hide the fragments of text which are not different in style from the surrounding text (like in the example above). The reason is that `*ngIf` can be only applied to a tag, and not to a text fragment, and `<span>` by itself does not alter the resulting display of the enclosed markup.

Conditional styles

It is also possible to specify the styles depending on condition (e.g. to show a status message in a red or green color depending on a status value). This is handled by the `[ngStyle]` directive. It again looks like a html attribute, its value now defines the condition for the style in the following format: `[ngStyle]="{'`*`property`*`' : `*`condition`*`}"`, where *`property`* refers to a CSS property like `color` or `padding`, and *`condition`* is an expression in TypeScript returning one of the possible property values.

This is rendered to show the name of the `x` object in green if it is good, and in black otherwise:

```
<span [ngStyle]="{'color': x.good ? 'green' : 'black'}">{{ x.name }}</span>
```

Note that we use single quotes to define string literals within a conditional style defined as html attribute value (i.e. inside the double quotes), this is a common practice in Angular where complicated expressions often appear within double quotes in html-like markup.

Summing up

To sum it up: to display anything on the component view it is necessary to

- define the data as the properties in the component class
- refer to this data directly or as parts of expressions in the placeholders within the component template
- style the display with the component style file

Exercise 3 – Angular Components and Forms

Custom components

Our own components are structured the same way as the application component. When you run `ng generate component `*`component-name`*, the result will be put in the directory named *`component-name`* and will include the set of the files described above (class, template, and style). In addition, the component class name will be included in the declarations section in the `@NgModule` decorator parameter. Note: if the component fails to render, it always makes sense to check if its class is really included in that section.

To render the component, its selector tag must be included in the template of some other (parent) component. Suppose we have `ChildComponent` defined as follows:

```
// child.component.ts
@Component[{
  selector: 'child-component',
…
}] export class ChildComponent { ... }
```

```
<!-- child.component.html -->
<div>I am the child component</div>
```

then we can use it in the parent component's template as follows:

```
<!-- parent.component.html -->
<div><span>Here goes the child component</span>
<child-component></child-component>
</div>
```

The result will be rendered as follows:

```
<div><span>Here goes the child component</span>
<div>I am the child component</div>
</div>
```

Input parameters for components

Often it is necessary to share data between parent and child component. The simplest case is when the data goes from the parent component to the child, e.g. when a child component is used to display an element of the array belonging to the parent component.

For this, it is necessary to

- declare input parameters in the child component class;
- declare the values to be passed to these parameters in the selector tag of the child component within the template of the parent component;

As a result, the values passed from the parent component will initialize the parameters of the child component when it is rendered.

To declare the input parameter in a component, it is necessary to precede its declaration in the body of the component class with a `@Input()` decorator.

```
// thing.component.ts
import { Input, ...} from '@angular/core';
@Component[{
  selector: 'app-thing',  // such names are created by ng generate component
…
}] export class ThingComponent {
   // defines the component input, which
   // comes from the parent component when this component is rendered
   // the name of this parameter will be specified in the template of
   // the parent component as [currentThing]="value" attribute of the
   // <app-thing> tag
   @Input() currentThing: Thing;
}
```

To pass the value of the input parameter from the parent component, it is necessary to specify it as the value of the `[parameter-name]` attribute of the child component tag in the template of the parent component: `[parameter-name]="value"`; here `value` can e.g. refer to the name of the property of the parent component, or to the current element in an array defined by `*ngFor`, or to a result of the function call or other expression.

Suppose we have a thing list component (a parent) which declares a thing array as its data (defining `thingList: Thing[]` property), and wants to render a list of things, when each thing is rendered by means of the thing component (a child). Then the template for the parent component will look as follows:

```
<!-- thinglist.component.html -->
Things:
<app-thing *ngFor="let thing of thingList" [currentThing]="thing">
</app-thing>
```

While rendering the child component, the `currentThing` property will be initialized with the value coming from the parent component. In this case, it will be initialized to the element of the `thingList` array for every instance of the child component.

```
<!-- thing.component.html -->
<div>
Thing: <span [ngStyle]="{'color' : currentThing.good ? 'green', 'black'}"
       >{{currentThing.name}}</span>
</div>
```

This all will be rendered as follows:

```
Things:
<div><span style="{color:green}">Thing: the first thing</span></div>
<div><span style="{color:black}">Thing: the second thing</span></div>
```

Forms in Angular

Angular forms is a very complex topic, here we will limit ourselves with only one way of building the forms (template-based forms), one way of dealing with form input values (two-way model binding), and only one type of validation (required fields); it will allow to complete the assignment.

To deal with template-based forms in Angular, it is necessary:

- to include the form markup in the component template
- to specify the connection between the form input elements and the property values defined in the component class (the model binding for the form).
- to declare the event handlers for form elements in the component template
- to implement these handlers in the component class

Suppose we would like to add new things to the list of things, specifying the name and the good status for a new thing. As we need to change the whole list by adding the elements, and not the specific thing, it makes sense to put this form into the thing list component.

First, we need to think how we should handle the form data in the component class. Usually adding new elements to the list by means of the form can be done as follows:

1. the component declares the property holding a new data item, this will serve as a model to bind the form elements to:

```
public newThing: Thing;
```

2. this property is initialized with a newly created empty object in the component constructor

```
this.newThing = { name: "", good: false };
```

3. then its attributes are filled in a form
4. on submitting the form, the reference to the value of this property (which is an object) is added to the list

```
this.thingList.push(this.newThing);
```

5. after that this property is re-initialized to refer to a newly created empty object:

```
this.newThing = { name: "", good: false };
```

This works, as JavaScript. like Java, supports reference semantics, every object variable is just a reference to its data, so reassigning a reference does not destroy the data, and it remains accessible by the other references. In our case, after reassigning the empty value to the `newThing` property, the added array element still refers to the filled object.

For now. we just need to define the property in the component class, and initialize it in the constructor (steps 1-2 above), the rest will be done later.

The next step is adding the form markup to the component template. It is done as follows:

```
<form #newThingForm="ngForm">
<!-- form elements -->
</form>
```

Here `newThingForm` defines the name of the *form object* which will allow us to refer to the form as a whole, this will be useful for the validation and will be shown later.

Now we need to define the form elements. We now will limit ourselves with only three types of form elements: a text input field (to enter the name), a checkbox (to enter the "good" status), and a button (to submit the form).

First, we need to specify the input field for the thing name:

```
<input name="name" id="name" type="text" placeholder="Enter thing name"
  required [(ngModel)]="newThing.name">
```

Here we see two differences from the pure html input field: the `required` property and the `[(ngModel)]` directive. The `required` property specified that this input field is required. It activates the default validator for the required fields, as a result, the form object (in our case, `newThingForm`) will hold the `invalid` status until the field is filled. We will query this status when dealing with the submit button below.

The double "banana-in-a-box" brackets around the `[(ngModel)]` directive define the **two-way model binding** for this form element. It means that

1. any changes made by the user with the contents of this input field are immediately reflected in the value of the bound property (in our case, `newThing.name`),
2. whereas any changes in the value of the bound property e.g. by assigning new values within the component class, are immediately reflected in the display of the form element.

As a result, when the user enters something in this field, the entered text is immediately assigned to the bound property. As a result, when the form is submitted, the bound property `newThing.name` reflects the data entered so far.

Now we need to define a checkbox to set the good status of the added thing. It is done in a similar way; the bound property is `newThing.good`:

```
Good: <input name="good" id="good" type="checkbox"
[(ngModel)]="newThing.good">
```

Now we need to define a submit button of a form.

```
<button type="submit" (click)="addThing()"
[disabled]="newThingForm.invalid">Add new thing</button>
```

Here we see two new elements: the `[disabled]` attribute and the `(click)` event handler definition.

1. The `[disabled]` attribute refers to a condition which, if true, makes the form element disabled. In our case, it refers to the `newThingForm` form object and is true when it is invalid i.e., when its required elements are not filled.
2. The `(`*`event-type`*`)="`*`handler`*`()"` syntax defines the **event handler** for the *`event-type`* events fired by this form element. Such handler is a method of the component class which is called when the event of the specific type occurs. The type of the event is specified within the parentheses, in our case it is a `click` event which is fired when the button is clicked. As a result, the `addThing()` method of the component class is declared to be called when the user clicks on this button.

Now we need to define the `addThing()` event handler for this form. As we can be sure that the newThing property reflects the data entered in a form (due to the two-way model binding doing its job), we can proceed with adding it to a list and assigning it to an empty object:

```
// thinglist.component.ts
addThing(): void {
     // add new element to the list,
     // assign new element to the empty element (so the form is cleared)
      this.thingList.push(this.newThing);
      this.newThing = {name: "", good: false};
    }
}
```

Here we can also check if the element exists in a list, if it exists, and we do not want to add duplicate elements, we have a form submission error which we need to handle. It can be done as follows:

1. we declare the error message property in the component class:

```
errorMessage: string = null;
```

2. we declare the placeholder for this message in the component template which shows it only if it is set

```
<!-- thinglist.component.html -->
<div *ngIf="errorMessage != null">{{message}}</div>
```

3. we set the error message property of the component class to some specific message in case of error, clear it (set to null) when the error condition is no longer applicable.

```
addThing(): void {
  if (this.thingList.find((t) => t.name == this.newThing.name) != null) {
     this.errorMessage =  `${this.newThing.name} already exists`);
  } else {
     // proceed with adding new element as above
  }
}
```

Sometimes forms are simpler. For example, in some cases only the button is necessary with a single handler of the (click) event. Suppose we want to change the value of the `good` property of the specific thing within the thing component. To do so, we can add the pair of buttons "make good" and "make bad" to such component and provide the corresponding handlers. Only one of these two buttons must be shown depending on the current value of this property.

In the template, we put two buttons with `*ngIf` attached to both and with the corresponding handlers:

```
<!-- thing.component.html -->
<form>
  <button *ngIf="!currentThing.good" (click)="makeGood()">make good</button>
  <button *ngIf="currentThing.good" (click)="makeBad()">make bad</button>
</form>
```

The handlers just change the value of `this.currentThing.good` correspondingly.

Output parameters for components

In some cases, it is also necessary to pass data from the child component back to the parent component. An example is the case when we need to inform the parent component that something happened within one of its children, together which the information about the affected child. This can allow e.g. to show the message "something happened with the thing x" above the list of things.

For this, it is necessary to use the mechanism of **event emitters** which allow to emit custom events in the child components which can be handled by the handler methods defined in the parent component. To do so, it is necessary to:

- declare the output parameters in the child component class as its properties which are the initialized instances of the `EventEmitter` class;
- when necessary, emit the events in the child component by calling the `emit()` method of the output parameter emitters and passing the data.
- declare the handlers of the emitted events in the selector tag of the child component within the template of the parent component;
- implement the handlers as the member functions of the parent component;

As a result, the values passed from the child component will be available to the custom event handlers in the parent component.

Suppose we would like to show the message "Thing x is made good" or "Thing x is made bad" above the list of things in the parent component whenever the value of `currentThing.good` is changed in one of the instances of the child component.

First, we need to define the output parameters in the child component. To do so, we need to precede their names with a `@Output()` decorator and made them the initialized instances of the `EventEmitter` class parameterized by the type of the data to be included with an event:

```
import { EventEmitter, Output, ...} from '@angular/core';
export class ThingComponent {
   // defines the component outputs, which
   // go from the child component up to the parent as custom events
   @Output() onThingMadeGood: : EventEmitter<Thing> = new EventEmitter();
   @Output() onThingMadeBad: : EventEmitter<Thing> = new EventEmitter();
}
```

Now we need to emit the events. To do so, it is necessary to call the `emit()` method of the emitter instances passing the value which we want to transfer to the parent as a parameter of these methods.

```
export class ThingComponent {
// ...
   makeGood(): void {  // e.g. the (click) handler of the "make good" button

    // change the value for the current thing and emit the event,
    // it will be captured in the thing list component
    // by the handler of the onThingMadeGood event

    this.currentThing.good = true;
    this.onThingMadeGood.emit(this.currentThing); // send the current thing
  }

  // the same for makeBad(), but call this.onThingMadeBad.emit()
}
```

Now we need to define the handlers for these events in the template of the parent component. Such handlers are specified similarly to the handlers of the standard UI events such as click: `(`*`outputParameterName`*`)="`*`handlerName`*`($event)"`. In our case, it can be done as follows:

```
<!-- thinglist.component.html -->
<app-thing *ngFor="let thing of thingList" [currentThing]="thing">
            (onThingMadeGood)="thingWasMadeGood($event)"
            (onThingMadeBad)=" thingWasMadeBad($event)"></app-thing>
```

Then we implement the handlers within the parent component, the event parameter for these handlers will hold the value passed from the child.

```
// thinglist.component.ts
export class ThingComponent {
  // a message to be set
  public message: string = null;
  // a handler for "made good" event, the message is set according to the
  // coming data
  public thingWasMadeGood($event: Thing) {
      let nameOfChangedThing = $event.name;
      this.message = `Thing $nameOfChangedThing was made good`;
  }
  // the same for thingWasMadeBad()
}
```

Now we can show the message in the template of the parent component if it is set:

```
<!-- thinglist.component.html -->
<div *ngIf="message != null">{{message}}</div>
<!-- the thing list as above -->
```

Dynamic rendering based on form values

Sometimes the changes introduced in a form have to be immediately reflected by the other elements on a page. This can be also handled by the two-way model binding as follows:

1. bind the property to a form element
2. use this property in the `ngIf*` condition or in a function returning the array to be used in the `*ngFor` directive.

As a result, the changes in the property will immediately affect the condition or the list and this will be immediately shown to the user.

Suppose we would like to filter the things in a list to match the string entered in an input field. This can be done as follows:

1. create the `filter` property in the component class;
2. bind this property to the input field as above using two-way binding;
3. use this property to filter the `thingList` in a function which returns the filtered list; this function can e.g. be defined as follows:

```
filteredList(): Thing[] {
    return this.thingList.filter((t) => this.filter == null ||
                          t.name.indexOf(this.filter) != -1);
}
```

4. use the call to this function as a list in the `*ngFor` directive:

```
*ngFor="let thing of filteredList()"
```

Dealing with list boxes

It is possible to change the contents of a list box or a dropdown box dynamically, again due to the two-way model binding. The template representation can be as follows

```
<select id="things" name="things" size="3" [(ngModel)]="thingToDelete">
    <option [value]="thing.name" *ngFor="let thing of thingList">
        {{thing.name}}
    </option>
</select>
```

Here the list box element has a two-way binding to the `thingToDelete` component property which is a selected thing in a list. The list elements are rendered from the `thingList` array using `*ngFor`, `[value]` here provides the value for every element which will be assigned to the `thingToDelete` attribute on selecting the element, here our things do not have unique ids, otherwise the id attribute would have been the good candidate for using with `[value]`. The text between `<option>` and `</option>` defines what is visible for a specific list element.

As a result, due to the two-way model binding the changes in `thingList` are immediately visible in a listbox, so if we delete the element from the array, it will be also deleted from the listbox, adding the element to the array also makes it added into listbox. On the other hand, changing `thingToDelete` in code (e.g. assigning a new name to it) leads to selecting the new element in a list. E.g. if we delete the selected element, we may want to select the previous element in a list, the deletion code can be as follows:

```
deleteThing(): void {

    // find the place for the selected thing in the thingList array
    let positionToDelete = this.thingList.indexOf(this.thingToDelete);

    // delete the element form the list, rendering the select element
    // immediately due to the two-way model binding
    this.thingList.splice(positionToDelete,1);

    // adjust the selected element to point to the previous one
    if (this.thingList.length == 0) this.thingToDelete = null;
    else if (positionToDelete != 0)
    this.thingToDelete = this.thingList[positionToDelete - 1];
}
```

Summing up

To deal with multiple components in an application it is necessary to

- create every component file by means of Angular CLI or manually
- refer to the selector tag of the child component in the parent component template: `<child></child>`
- pass the data from parent to child via the values of the input parameters of the child component specified in the `[`*`input-parameter`*`]="`*`value`*`"` attributes of its selector tag
- pass the data from child to parent via the values attached to the custom events emitted from the child component via its output parameter emitters, and handled in the parent components in the handlers of these events defined by means of the handler specifications `(`*`output-parameter`*`)="`*`handler`*`($event)"` in the selector tag of the child component

To deal with the forms in the application, it is necessary to

- define the property to hold the form data in the component class, initialize it in the constructor of that class
- create the form template, specifying the model binding for the form controls (e.g. two-way binding `[(ngModel)]="`*`boundProperty`*`"`) and the handlers for its events `(`*`event-type`*`)="handler()"`
- implement the event handlers, in the handler of the form submission deal with the bound property as it contains the data which was changed by the user

Exercise 4 – Angular Routing and Services

Routing in Angular

Until now, all interaction within the Angular application used a single URL like `http://`*`host`*`/app/` (or `http://`*`host`*`/` if it is deployed to the document root, we will use such URLs below). In some cases, it is more convenient to allow access to different sections of the site with different URLs such as `http://`*`host`*`/products` or `http://`*`host`*`/product/`*`product-code`*. The advantages are as follows:

- such site is better suited to being indexed by the search engines
- separate URLs can be bookmarked, shared on Facebook etc.
- URLs can contain the data which is passed between components such as the product code above

- components can render the independent views loaded one after another into the same space (e.g. after being selected from a menu), as they are not always suited to be the part of the component hierarchy.

Allowing such access is supported by **routing**. Angular router is a component which processes request URLs and renders different components depending on the information contained in these URLs. Angular supports two kinds of routing: path-based and hash-based.

1. With path-based routing, the route URLs are formed as regular URL paths: `http://host/path/to/resource`; such routing requires support implemented on a hosting server, as these paths first come to the server, and it must understand that they have to be redirected to the Angular application. Otherwise it will return 404 status because such resources do not exist there.
2. With hash-based routing, the route URLs are formed like this: `http://host/#/path/to/resource`. The difference is in using a hash symbol in the URL. As, per the URL standard, the hash is a separator between the page address and the *page anchor* (the identifier of a place within a page), all such requests come to the server to ask for the same network resource – the root page of the Angular application `http://host/`. The anchor part `/path/to/resource` is ignored by the server and passed "as is" to the application. After coming there, the anchor part is handled by the Angular router which renders different components based on the information contained in that part. As a result, no server support is needed, as the server sees all the requests asking for a single existing resource. We will use hash-based routing in this assignment.

To add routing support, it is necessary to specify the router module in the `app.module.ts` as follows:

```
import { RouterModule } from '@angular/router';
import { ROUTES } from './app.routes';
@NgModule({ ...
  imports: [
    RouterModule.forRoot(ROUTES, { useHash: true })
  ] ... }) export class AppModule { }
```

This code adds a router module to the imports list of the application module, it accepts the array of routes (the `ROUTES` parameter coming from the file `app.routes.ts`) and a parameter indicating that we need hash-based routing. Before specifying the `ROUTES` array in `app.routes.ts`, we need to know which routes we need. We start with the simplest case of the routes which do not include parameters.

Suppose for our thing list application we plan to have a menu on top of our page, which stays there all the time, and provides a set of links. Clicking on links renders different page fragments (served by different components) in the space below the menu. For now, we will need two routes: one for the main page (the thing list component) and another for the contacts page of the application. Both must be accessible from the menu; **they do not form a hierarchy**.

The list of routes to be imported by the router component is specified in the file `app.routes.ts`. The routes data is requested by the router as an array of route objects, a route object has at least two attributes: a route path and a component which is responsible for handling the route; such component is rendered by the router when the browser sends a request to the route path.

```
export const ROUTES: Routes = [
  { path: '', component: ThinglistComponent },
  { path: 'contacts', component: ContactsComponent }
];
```

The first route in a list is a root route which indicates the component which is rendered when the browser sends the request to `http://`*`host`*`/#/`. The second route specifies the component to be rendered when the request comes for `http://`*`host`*`/#/contacts`.

Routing usually does not affect the whole page occupied by the application. For example, in our case the menu is not affected by routing, it stays on top regardless of the requested route. This is handled as follows:

1. we decide which component will hold *the router outlet* i.e. the space where the router will render the components based on requested routes.
2. we put the `<router-outlet></router-outlet>` tag into the template of such component
3. now navigating to the router links changes the display only inside the `<router-outlet>` tag.

We put the `<router-outlet></router-outlet>` tag into the template of the application component `app.component.html`, the menu will go into the same template above the tag.

```
<!-- here will go the menu -->
<router-outlet></router-outlet>
```

Now we need to define the menu. It should contain the links to the root and contacts routes. The problem is that using standard HTML links: `<a href="/#/contacts">Contacts</a>` will not work well, as following them reloads the whole application page – including the menu. Angular provides the solution to that problem by using the `routerLink` attribute of the `<a>` tag. Following such link instructs the router just to render the component within the router outlet, without reloading the page. Now the menu in `app.component.html` will look as follows:

```
<a href="" routerLink="/">Thing list</a> |
<a href="" routerLink="/contacts">Contacts</a>
<router-outlet></router-outlet>
```

Note that the hash symbols are not specified in relative `routerLink` URLs, they will be automatically added by the router. Still, if you need to bookmark or share a specific router URL, or to link to it from another place, that URL must include the hash:

```
<!-- a page hosted by a link aggregator somewhere -->
<a href="http://host/#/contacts">Thing manager contacts</a>
```

Parameterized routes

If we need to pass the data to the components within the route URLs, e.g. to use different URLs for the detailed display of different things, we need to *parameterize the routes* as follows:

1. in the ROUTES array to be passed to the application module, the route path should include the parameter name preceded by the colon:

```
export const ROUTES: Routes = [
  // shown when host/#/thing/name is accessed, parameterized by name
  { path: 'thing/:name', component: ThingDetailComponent }
];
```

2. In the router link, the parameter value is specified as a part of the URL without the colon:

```
<div *ngFor="let currentThing of thingList">
    <!-- sends the value of the name attribute to the route -->
    <a href="" routerLink="/thing/{{currentThing.name}}">Thing detail</a>
</div>
```

3. In the component connected to the route, it is necessary to use the ActivatedRoute object which is injected into its constructor (we will discuss the dependency injection in the next section, for now it is enough just to know that this object will be always available to the component).

```
import { ActivatedRoute } from '@angular/router';
export class ThingDetailComponent {
  constructor(
  // inject the activated route object as a route property
    private route: ActivatedRoute
  ) {
    // get the value of the name route parameter
    let name = this.route.snapshot.paramMap.get('name');
  }
}
```

In the code above, the `name` variable will get the value of `'myThing'` if the requested route was as follows: `http://host/#/thing/myThing`.

Until now, we mostly passed the data between components belonging to the same component hierarchy i.e. which can communicate via input and output parameters. By using parameterized routes, it is possible to share data between components not belonging to such hierarchy. All we need to pass the data is the route URL, not the component tag. In the code above, both thing list and thing detail components render their views within the same router outlet space, they are not in the same hierarchy.

Dependency injection and services

Route parameters are the limited means of data sharing between components as the URLs are suitable to pass simple (scalar) parameters, not complex objects. Moreover, they do not allow to share *behavior*. All this is better done by introducing **services**.

A service is a class, the single instance of which is shared between components together with its dependencies. Such sharing makes use of the **dependency injection** paradigm. Let us look at it in more detail.

Dependency injection is a resource and behavior sharing paradigm when the instance of some class together with all its dependencies is made available to the instance of the other (target) class simply by being declared as a member of that class or as a parameter of its constructor. It is said that such instance is *injected* into another (target) instance. The availability and the proper state of the injected instances rely on some "magic" provided by the runtime environment, such environment (a dependency injection container) creates or reuses the instances of the injected classes when it encounters the appropriate declarations in the code of target classes, such instances are provided to the target instances to serve as their members at runtime.

In Angular, the runtime environment is provided by the Angular runtime. With the support of the runtime, the service instance is injected into the instances of the component or the other service if it is declared as *a member-defining parameter of its constructor*:

```
class X ( public theService: ServiceClass ) {
   // theService will be magically available at runtime as a class member
}
```

The injection is made together with all dependencies, so e.g. if the service `A` has another service `B` injected into it, its injection into the component `C` leads to the runtime availability of both its own instance and the instance of the service `B`.

To inform Angular runtime that the instances of the class can be injected into the instances of other classes, it name has to be preceded with `@Injectable` decorator:

```
@Injectable
export class ThingListService { }
```

Here we declare that the instances of a `ThingListService` can be injected into the instances of other classes. In particular, its instances can be injected into components and all its resources can be shared among them.

Suppose we want to share our thing list among all the components in the application, even if they do not belong to the same hierarchy, like thing list and thing detail components. It is possible to implement it as follows:

1. define a service class (in `thinglist.service.ts`) as described above
2. list that class in the providers section of the application module, such section holds all injectable classes which can be used in the module:

```
import {ThingListService} from "./thinglist/thinglist.service";
@NgModule({
  // the providers section lists services
  // which can be injected into components
  providers: [ThingListService],
})
export class AppModule { }
```

3. make `thingList` a property of the service class, initialize it in its constructor (e.g. from the JSON file like it is shown in the assignment itself)

```
@Injectable
export class ThingListService {
   private thingList: Thing[];
   constructor() {
      this.thingList = globalThingList;  // e.g. from JSON
   }
}
```

4. create the set of service members providing common thing list operations like `getThingList()`, `filterThingList()`, `addThing()`, `deleteThing()` etc.
5. declare that the service instance should be injected in the target component classes by specifying it as a member-defining parameter of their constructors:

```
export class ThingListComponent {
   constructor( public thingListService: ThingListService ) {}
}

export class ThingDetailComponent {
   constructor( public thingListService: ThingListService ) {}
}
```

6. refactor the code to use the service and not `thingList` directly:

```
// in ThingListComponent
addThing(): void {  // click handler for the submit button
   this.thingListService.addThing(this.newThing);
}
// in ThingDetailComponent
constructor(public thingListService: ThingListService ) {
   // get the name from the route parameter as above
   this.currentThing = this.thingListService.getThingByName(name);
}
```

It is important to understand that all component instances will share the same instance of `thingList`, as the same instance of `ThingListService` (containing a single `thingList`) is injected in all these instances, this is guaranteed by the Angular runtime.

Summing up

Routes allow to address parts of the application with different URLs. To use them it is necessary

1. to add the `RouterModule` module to the "imports" section of the application module parameter parameterized with the route specification which maps route paths to components
2. to add the `<router-outlet>` tag to the template of the component which owns the space where the routed components are rendered
3. to put links containing the `routerLink` attribute where we need to navigate to the routes, so they are rendered to the router outlet

Parameterized routes allow to pass data between components not belonging to the same component hierarchy. To do this, it is necessary

1. to add `:name` parameters to the router paths specified in the route specification
2. to put values into URLs specified in the `routerLink` links

3. to get the transferred values in the routed components by means of the injected `ActivatedRoute` object.

Services allow to share data and behavior among components. The services make use of the dependency injection paradigm, so their instances are made available to the components by the Angular runtime by declaring them as parameters of their constructors. To implement the service, it is necessary

1. to prepend its name with the `@Injectable` decorator
2. to include it in the "providers" section of the application module parameter.

To use the service in the component, it is enough to add it to the parameter list of its constructor as a member-defining parameter.

Exercise 5 – REST Client in Angular

Observables and asynchronous calls

Asynchronous programming in Angular (which e.g. includes handling HTTP calls) does not use promises, instead, it uses another approach called Observables. Using Observables relies on a set of classes provided by the RxJS library (where `Observable` is a central class). Their usage is similar to promises to some degree, but the details are different. Here only the very basics will be covered, to the degree of making possible to use the HTTP client library.

In general, the approach is similar to promises: we perform an asynchronous call which does not return immediately, instead, it allows us to register callbacks to be called on successful or unsuccessful completion of the call (i.e. when the data is returned back or when the error is returned instead). The main difference is that, while promises allow for a single piece of data to be handled once (they are either accepted or rejected), Observables provide a stream of data to which the clients can *subscribe*, before that the data can be transformed, or some additional actions can be performed, all the transformations form the chain (a pipe) where the result of the previous call in a chain are provided to the next call.

Observable is a parameterized class; the parameter is a type of data to be available to their subscribers: `Observable<Thing>` will allow to subscribe to a stream of Things.
Suppose we have a function `getThingAsync(): Observable<Thing>` which is called asynchronously returning a stream of `Things` to which we can subscribe. Performing the simplest asynchronous processing by means of such function is done as follows:

1. call the function returning an `Observable` and chain the call of the `subscribe()` method to the function call: `getThingAsync().subscribe(...)`. This is possible because `subscribe()` is a method of `Observable`.
2. pass the subscribing callback to the call of `subscribe()`:

```
getThingAsync().subscribe((param: Thing) => {
   // callback code, param is available
});
```

The code following the call to `subscribe()` will be executed immediately after the call, it is not guaranteed that it happens before the callback starts executing.

3. in the callback, handle the `param` parameter of the parameter type of the `Observable` (`Thing` in our example) which will be available when the call returns the data. Until then, the callback will not be executed as it will wait for the data to arrive:

```
// somewhere where we have a currentThing property
getThingAsync().subscribe((param: Thing) => {
   // callback code, param is available
   this.currentThing = param; // set when the data arrives
});
```

It is also possible to transform the data before finally subscribing to it. It is done by prepending the filtering calls to the `subscribe()` call in a chain. Usually it is done by the nested pair of `pipe()` and `map()` calls:

```
import {map} from "rxjs/operators";
getThingAsync().pipe(map((t: Thing) => t.name ))
   .subscribe((param: string) => { });
```

Here we converted the `Thing` into its name and subscribed to the result. Note that the map's callback should return the value of the specific type e.g. `Thing` or `string`, but the result of the call to `map()` in the chain will be the `Observable` parameterized with this type: we return `string` in the above example, but the returned value of the `pipe(map())` will be `Observable<string>`.

It is also possible to return `Observable` from our own functions allowing others to subscribe to their results. For example, the result of the above conversion can be returned from the function, then it will be possible to subscribe to it:

```
import {Observable} from "rxjs";
import {map} from "rxjs/operators";
function getThingNameAsync() : Observable<string> {
      return getThingAsync().pipe(map((t: Thing) => t.name ));
}
// somewhere where we have a name property
getThingNameAsync().subscribe((name: string) => {
   this.name = name;  // set when the data arrives
});
```

Up to now we dealt with the successful completion of the asynchronous calls. If we want to handle errors, it is necessary to call `subscribe()` with two parameters, where the second accepts a callback which will be called in the case of errors:

```
getThingAsync().subscribe(
(thing: Thing) => {
   this.currentThing = thing;  // set when the data arrives
},
(error) => {
   // handle errors here
});
```

All the conversions happening in a chain do not affect the error handling, so if `getThingAsync()` raises an error, this error reaches the second callback of `subscribe()` through all the prior calls to `pipe(map())` etc., so the errors handling of e.g. `getThingNameAsync()` will be the same.

Using HttpClient library

Implementing HTTP client in Angular relies on `HttpClient` library. Such library provides an injectable `HttpClient` service with member functions corresponding to different HTTP methods: `get()`, `post()` etc. Every such method returns an `Observable`, so it is necessary to subscribe to its results to get the data from the call.

Suppose we have a REST service available on localhost at port 2000 which provides the operations with things. Performing calls to this service in a component requires the following steps.

1. make the service instance injected into the component:

```
import { HttpClient } from "@angular/common/http";
export class ThingListComponent {
   // the service is injected to allow making HTTP calls
   constructor(private http: HttpClient) {
   }
   // now we can call http's methods
}
```

2. call the methods of this injected instance and subscribe to their results. For now, we will call the `get()` method performing a GET request to the REST service. The first parameter of this call is an URL which is provided by the REST service for the corresponding call. Suppose our thing service provides a GET entry point returning all available things accessible by the `http://localhost:2000/things` URL. Then the call to this entry point will look like this:

```
// this is the base url of the REST service
const baseURL = "http://localhost:2000";

constructor(private http: HttpClient) {
   this.http.get(baseURL + "/things").subscribe(...);
   // get() returns Observable, so we can subscribe to it
}
```

3. subscribe to the results of the call and process the received data. By default, `get()` and the other calls in the library return `Observable` parameterized by the type of the expected data, which is transparently converted from the JSON coming from a

call. As our REST call returns an array of `Thing`s, subscribing to such call makes possible to use the callback parameter which is of a type of the expected result, such as `Thing[]` in our case:

```
export class ThingListComponent {
thingList: Thing[] = null;
constructor(private http: HttpClient) {
   this.http.get(baseURL + "/things").subscribe((res: Thing[]) => {
      this.thingList = res;
   });
}
```

In the above callback we set the property of the component to the results returned from the call after the data comes from it asynchronously. This is a common practice in Angular. *Instead of directly filling the property values exactly when we want it, so these changes are reflected on display, we got these values filled for us in the callbacks of the asynchronous calls, and make the display reflect the changes in these values by using `*ngIf` in templates referring to the values which are the subject to change.*

Let us look at our `thingList` property. Now we do not know exactly when it will be filled, as the call to `get()` receives the data asynchronously, i.e. at some unknown point of time. Prior to that it will be equal to null, as we initialized it this way. The solution here is to include the check if the property value is not equal to null in the template:

```
<div *ngIf="thingList != null">
  <div *ngFor="let thing of thingList"> ... </div>
</div>
```

Now what we did relies on the behavior of the Angular runtime which refreshes the display from time to time checking if anything has changed in the component properties and re-evaluates the conditions in the templates in a process. So, initially the contents of the conditional `<div>` in the above template will not be rendered as the `thingList` property is not set yet. But at some later time, when the asynchronous call completes, and the callback is executed, it will set the value of the property, one of the refreshes will re-evaluate the condition to true, and the contents of the conditional `<div>` will be rendered, showing the thing list. *All this will happen without any explicit action to show the data.*

This is ***the most important point*** *in dealing with the asynchronous calls in Angular, so it is worth repeating:*

- *we set the values of the component properties to the results of the asynchronous calls in their subscribe callbacks.*
- *we include the values of these properties in the conditional statements in the component templates which allow to render them only when they are set*
- *Angular does the rest.*

Now we can look at the other calls. Suppose that in addition to the GET call, our REST service provides:

1. the POST call to add a new thing, which expects the thing to be added as a body of the request and returns JSON in the body with the added thing.
2. the PATCH call to change the "good" property of the specific thing, it also expects the changed thing to be in the body, but it also expects the name of the thing to change as the URL parameter, and returns JSON in the body with the updated thing.
3. the DELETE call to delete the thing, which expects only the URL parameter, and returns the JSON in the body with the deleted thing.

The POST call corresponds to the `post()` method of the `HttpClient`. As the REST call returns the added thing in a body, the `post()` method returns an `Observable` holding the added thing. It accepts the second parameter which is converted to JSON and goes as a body to the REST call. It is called as follows:

```
// thing: Thing holds a thing to add
this.http.post(baseURL + "/things", thing)
    .subscribe((added: Thing) => {
       this.message = `${added.name} was added`;
    });
```

Here the `message` property of the component can be shown on the template when the call completes as follows:

```
<div *ngIf="message != null">{{message}}</div>
```

The PATCH and DELETE calls correspond, again, to the `patch()` and `delete()` methods of HttpClient. They are called as follows:

```
// it is enough just to pass an updated property
this.http.patch(baseURL + "/things/" + thing.name, { good: true })
    .subscribe((updated: Thing) => {
       this.message = `${updated.name} was changed to good`;
    });

this.http.delete(baseURL + "/things/" + thing.name)
    .subscribe((deleted: Thing) => {
       this.message = `${deleted.name} was deleted`;
    });
```

Now we can look at handling the errors. Suppose our service also provides the GET call with an URL parameter to get thing by name, which returns an error with the status of 401 if the thing cannot be found, and includes the object with a `message` property in the body of the response in this case. `HttpClient` throws an error if the response comes with the status other than "success" (200). This error can be handled by the second callback of the `subscribe()` as follows:

```
this.http.get(baseURL + "/things/" + name)
   .subscribe((thing: Thing) => {
      // positive result callback
      // set the currentThing attribute to the obtained thing
      this.currentThing = thing;
   },
   (error) => {  // error callback
      // in case of error set the component attribute to the message
      // obtained from the server
      this.errorMessage = error.error.message;
   });
```

Here `error.error` refers to the object returned in a body. The `errorMessage` can be rendered on the template as before:

```
<div *ngIf=" errorMessage != null">Error: {{ errorMessage }}</div>
```

Incorporating HttpClient calls in a service

As a final touch, let us look how we can incorporate our `HttpClient` calls in a service. Such service will inject the `HttpClient` object:

```
@Injectable
export class ThingListService {
   constructor(private http: HttpClient) {}
}
```

Then it will implement a set of methods corresponding to the common tasks specific to our application, these methods will in turn call the methods of the `http` object, possibly convert their results, and return them as Observables, so the others (e.g. components) can subscribe to them. The errors have to be also thrown further, so they can be processed later.

```
getAllThings(): Observable<Thing[]> {
   return this.http.get(baseURL + "/things");
}
```

It is an important point: we never try to hide the asynchronous nature of the external calls in such services, we only hide the internals of making HTTP calls. The burden of handling the results of the calls in callbacks still lies on the clients of this service such as components. Only this way, we can make the components handle the property-setting machinery described above:

```
export class ThingListComponent {
thingList: Thing[] = null;
constructor(private thingListService: ThingListService) {
   this.thingListService.getAllThings().subscribe((t: Thing[]) => {
      this.thingList = t;
   });
}
```

Summing up

Asynchronous data processing in Angular relies on Observables. To make use with the function returning the `Observable` it is necessary:

1. to make call to this function and call `subscribe()` from its result (to subscribe to the call);
2. to handle the correct outcome in the callback passed as the first parameter to `subscribe()`, the parameter of this callback holds the data returned from the call. The callback will be called at some later point of time when the asynchronous call completes.
3. to handle the error in the callback passed as the second parameter to `subscribe()`.

Making HTTP calls from Angular relies on the `HttpClient` library. To make use of this library it is necessary:

1. to inject `HttpClient` service object into the component or the service;
2. to call the methods of this object corresponding to the HTTP methods: `get()`, `post()` etc;
3. to subscribe to the calls and handle the received data in the subscribe success callbacks or handle the errors in the error callbacks.

To make the received data shown by the component it is necessary

1. to define the properties to hold the data in the component
2. to call the methods of the `HttpClient` object in the component and get the data as parameters in the subscribe callbacks;
3. to set the component properties in these callbacks to the received data;
4. in the templates, use these properties in the conditional tags so they are rendered only when they are set.