Created by: Zoltán Richárd Jánki
Date: 09.05.2018

Angular 2+ - Data Binding & Services - Syllabus

Data-binding

Introduction: In the last practice, we have talked about the MVC design pattern. Since we divide the application following the MVC pattern, we have to find a solution for communicating among the different parts. The Data-binding is the solution for this communication in the Angular frameworks.

Types of data-binding: There are two types of data-binding: One-way data-binding and Two-way data-binding.

- The **One-way data-binding** makes a content of a variable visible in the View part. With the solution we cannot make any modifications on the content of the variable, we can just make it visible in the browser.

  In the source code: define a public variable in a component and use its name in the template (HTML) of the component with the following syntax: {{ variableName }}. See the example below!

main.component.ts

```
export class MainComponent implements OnInit {

      name: string = 'something';

      ...

      ngOnInit() { }


}
```

main.component.html

```
<section>{{name}}</section>
```

- The **Two-way data-binding** makes a content of a variable visible, but at the same time it can be also modified by some user-interactions. A typical usage of this type is in the forms (e.g.: registration page). This is the case when the user gives some input that we would like to use in our business logic.

  In the source code: define a public variable in a component and use its name in the template (HTML) of the component with the **`[(ngModel)]`** directive. Do not forget that using the **ngModel** requires the import of **FormsModule** in the **app.module.ts**. See the example below!

app.module.ts

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

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

Created by: Zoltán Richárd Jánki
Date: 09.05.2018

main.component.ts

```
export class MainComponent implements OnInit {
      url: string;

      ngOnInit() { }
}
```

main.component.html

```
<input [(ngModel)]="url">
```

Services

Following the Angular conventions, the Components should not fetch or save data directly. Their main is tasks are the presentation of data and delegation of data access to a service.
The services are kind of components that can be shared with other components. This sharing mechanism is handled by the Angular Dependency Injection (DI). A service is always injected into one or more components' constructor.
The Dependency Injection is a way to create objects that depend upon other objects. The Services are always signed as injectables with **`@Injectable()`** decorator. The easiest way to create a service is generating one with the help of Angular CLI. Use the following command in terminal in your project folder in order to create a Service class in the utils folder!

```
$ ng generate service utils/node
```

The above-mentioned command generates a new services called **NodeService** that can be found in the **`src/app/utils/node.service.ts`** path. There is also a generated *node.service.spec.ts* file that is responsible for testing.
As you open the **node.service.ts** file, you can find the **`@Injectable()`** decorator in it which means that it is now a dependency. If you also open the **app.module.ts** file, you can see that this Service class was added to the **`Providers[]`** array. This tells the injector how to create the Service. Without a provider, the injector would not know that it is responsible for injecting the service. Since it is added to our *app.module.ts*, this Service will be available for all of the components.
Now comes the injection! We have to create a new instance of the service in one of our component. Since it is a dependency, we do not have to care about the parameter list of its constructor. Let's use it in the *MainComponent*!

main.component.ts

```
import { NodeService } from '../utils/node.service';

export class MainComponent implements OnInit {

      constructor(private nodeService: NodeService) { }

      ngOnInit() { }

}
```

Now, we can call the methods of the service. It is injected into our *MainComponent*. Let's add the following code to the **NodeService**!

Created by: Zoltán Richárd Jánki
Date: 09.05.2018

node.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class NodeService {

  constructor(private http: HttpClient) { }

  getGreeting(): Observable<any> {
    return this.http.get('http://localhost:5000/rest/user/greeting');
  }

  registerUser(username: string, password: string): Observable<any> {
    let body = new URLSearchParams();
    body.set('username', username);
    body.set('password', password);
    let options = {
      headers: new HttpHeaders().set('Content-Type',
      'application/x-www-form-urlencoded')
    };
    return this.http.post(this.storageService.get('url') +
    '/rest/user/register',
    body.toString(), options);
  }

}
```

The **`getGreeting()`** method is responsible for sending an HTTP GET Request to our NodeJS server. It returns with an **`Observable<any>`**. The Observable objects are for handling the asynchronous operations. They are very similar to the Promises. The differences between them are written in the presentation of this practice!

So, when we call the **`getGreeting()`** method from our component, the component receives a response object with the type **`Observable<any>`**. Let's see, how can we catch and handle the response in the responsible component!

main.component.ts

```
export class MainComponent implements OnInit {
  ...
  greet() {
      this.nodeService.getGreeting().subscribe(data => {
            console.log(data);
      }, error => {
            console.log(error);
      });
  }
  ...
}
```

Created by: Zoltán Richárd Jánki
Date: 09.05.2018

In the code above, the **`greet()`** method contains the call of the **`greeting()`** from the NodeService. This is an asynchronous operation, so we have to use the **`(response => {});`** syntax. Since the greeting() method returns with an Observable, we have subscribe for this event with the **`.subscribe()`** method. The body of this method can be found after the **`=>`** sign. It is triggered only if the response has arrived. The response will be in the **`data`** object. ***(The data is just a name, it could be anything else, it does not matter!!!)*** If the response (**`data`**) has arrived, - and it is not an error - we can work with its content. Now we just make a simple console in order to see its whole content. The results **`console.log()`** statements can be seen in the browser's Inspect part. ***(Right click somewhere in the browser --> Inspect --> Choose Console)*** If the response is an error, then not the **`data`** part is triggered, but the **`error`**.

NOTE!
The following code is an anti-pattern and does not work well!

main.component.ts

```
export class MainComponent implements OnInit {
  dataToHandle: any;
  greet() {
      this.nodeService.getGreeting().subscribe(data => {
            this.dataToHandle = data;
      }, error => {
            console.log(error);
      });
      console.log(this.dataToHandle);
  }
  ...
}
```

With this code the problem is with the **`this.dataToHandle`** object. In this case, this undermost **`console.log()`** will return with an **`undefined`** value!
The problem is that the Service method is asynchronous, so it is executed in the background in order not to freeze the whole browser. Since it is asynchronous and the undermost **`console.log()`** is synchronous, the log returns with an undefined value, **because at the time when the undermost `console.log()` is executed, the `data` is not received from the Service. So, the assignment is not executed as well!**

Knowing this, when you would like to call two service methods after each other and you also would like to use the response of the first async method in the second async method, it is important to embed the two methods into each other. See the following example!

```
ngOnInit() {
  this.myService.method1().subscribe(data => {
    console.log(data);
    this.myService.method2(data).subscribe(data2 => {
      console.log(data2);
    });
  });
}
```

In this way we can use the received data object in the other method call as a parameter because it has a value at this point. If we put it outside the first async method's body, the two methods would run independently and there would be no guarantee to have any non-undefined value.
