COMP721 Web Development

AUT UNIVERSITY

Week 10: Introduction to JavaScript Framework Angular

Note: what we have studied of AJAX Event-Driven model is essential in understanding Angular!

Review of Lecture 9 (XML/JSON and Ajax interaction cycle)

- XML

Well-formed?
Valid?

```
<?xml version="1.0" encoding="UTF-8"?>
<Persons>
    <Person>
        <Name>
            <First>Thomas</First>
            <Last>Atkins</Last>
        </Name>
        <Age>30</Age>
    </Person>
    <Person>
        <Name>
            <First>Sachin</First>
            <Last>Tendulkar</Last>
        </Name>
        <Age>38</Age>
    </Person>
</Persons>
```

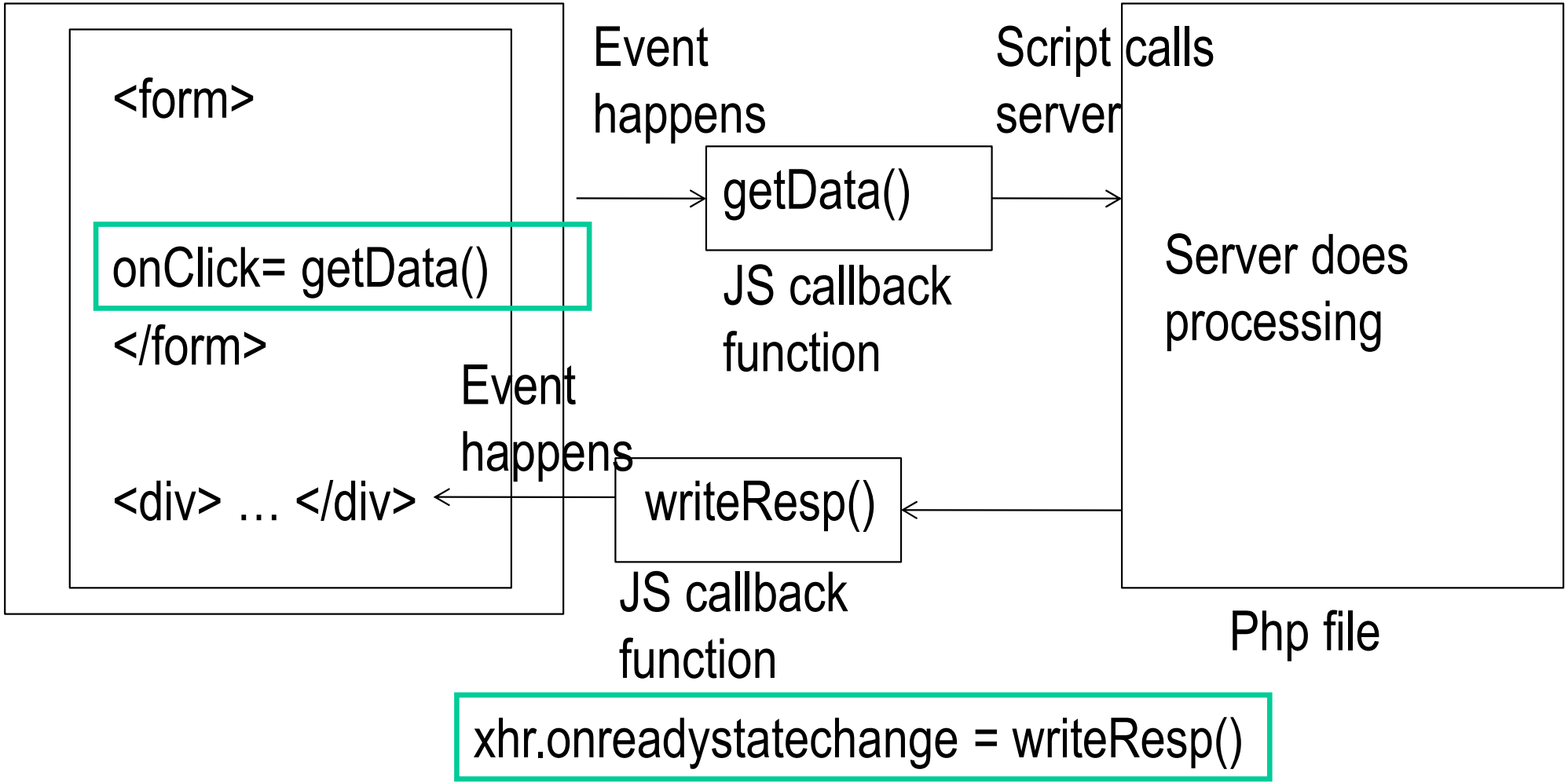
Review of Lecture 9

- Person DTD

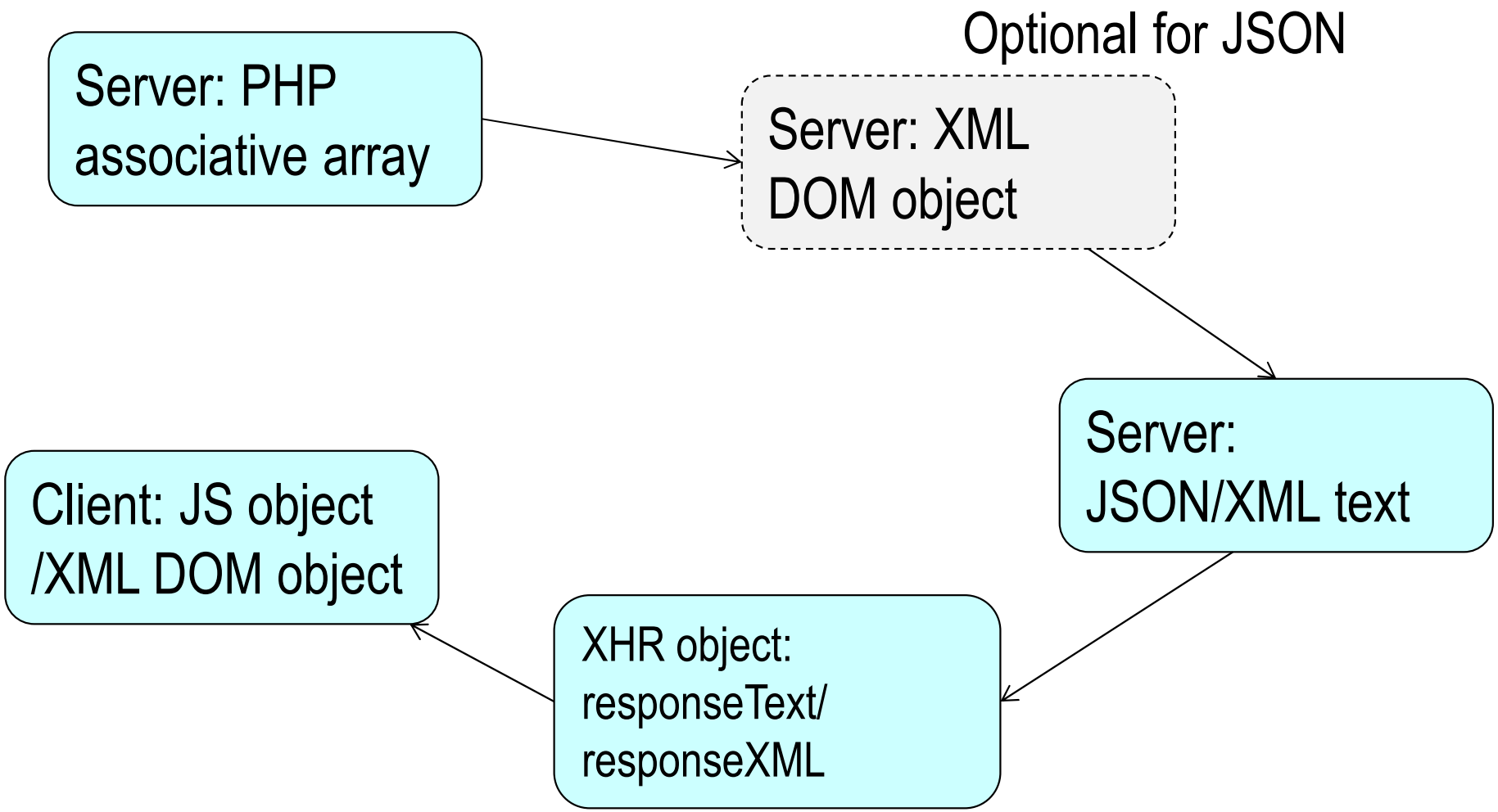
```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Persons ((Person+))>
<!ELEMENT Person ((Name, Age))>
<!ELEMENT Name ((First, Last))>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Age (#PCDATA)>
```

Review of Lecture 9

- Client-Server Ajax Interaction Cycle



Data Representation for Shopping Cart



XHR Object: The readyState Property

- This is a property of an XHR object. The possible values are
  - 0 – uninitialized
  - 1 – opened
  - 2 – send() has been called, and headers and status are available
  - 3 – receiving
  - 4 – loaded
- When an XHR object is created, its readyState property has the value 0. As processing continues, the property will actually take on all the values 0,1,2,3,4 in succession. Each time it changes, the onreadystatechange event fires, so the call-back function will generally be called 4 times before the value reaches 4 (ie, the value "loaded") .

XHR GET & POST

- GET: encode the query string values
  - ☐ `xhr.open("GET", "ManageCart.php?“+"book=" + encodeURIComponent(aBook));`
  - ☐ `xhr.send(null);`
- POST: encode the request body
  - ☐ `var requestbody =“book="+encodeURIComponent(aBook);`
  - ☐ `xhr.open("POST", dataSource, true);`
  - ☐ `xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");`
  - ☐ `xhr.send(requestbody);`

URL Encoding

- URL Encoding Reference (partial)

ASCII Character	URL-encoding
space	%20
!	%21
"	%22
#	%23
$	%24
%	%25

```
var requestbody
="name="+encodeURIComponent(
aName)+"
&pwd="+encodeURIComponent(a
Pwd);
```

- Javascript function: encodeURIComponent(string); the return is an encoded URI

Agenda

- Benefits of using frameworks
- Angular key features and architecture
- The root module
- Angular Components
- Angular Services (optional)

The advantages of using JS frameworks

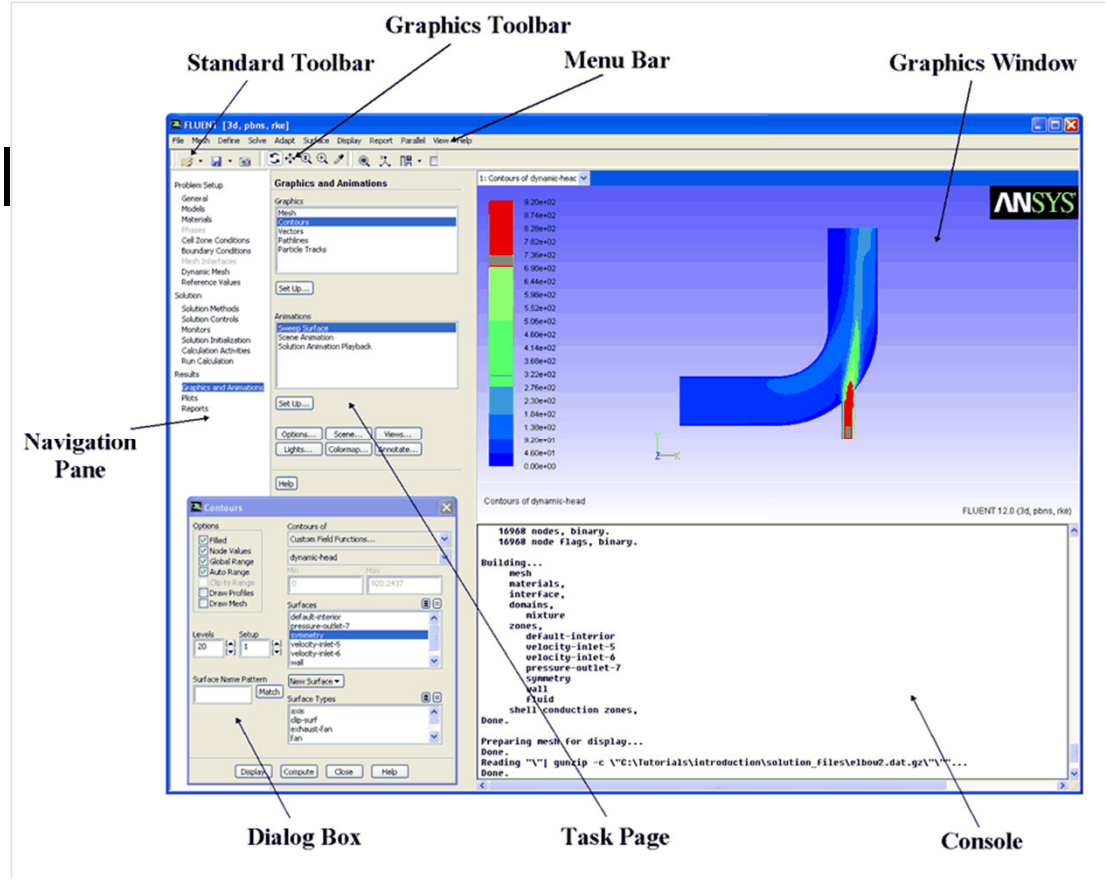
- **Efficiency**—projects that used to take months and hundreds of lines of code now can be achieved much faster with well-structured prebuilt patterns and functions.

- **Safety**—top javascript frameworks have firm security arrangements and are supported by large communities where members and users also act as testers.

- **Cost**—most frameworks are open source and free. Since they help programmers to build custom solutions faster, the ultimate price for web app will be lower.

What can be improved to the Ajax programming model?

- DOM is low-level fine-grained mechanism/API – reusability is low
- Can we treat the webpage as a desktop GUI?
- We also need an OO framework to manage large applications
- This is what Angular wants to achieve…



About Angular

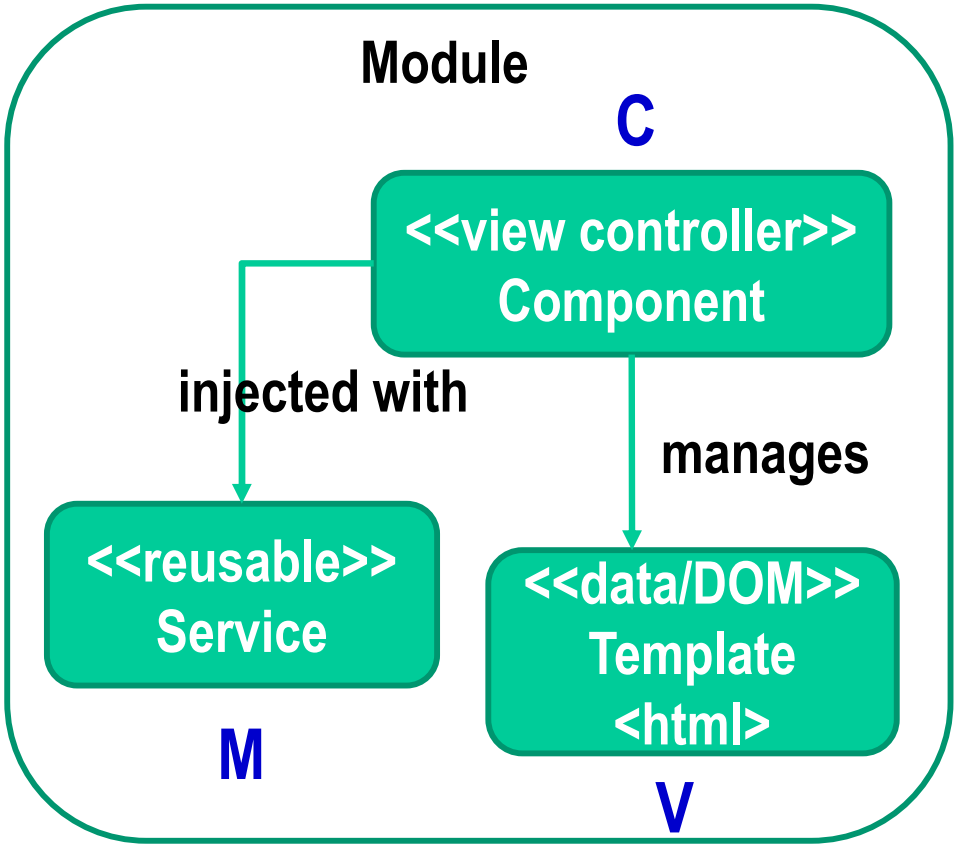
- Previous version: Angular.js
- Current version Angular 2 and above – released in 2016 (built on TypeScript: a superset of JS)
- Key features
  - two-way data binding: Two-way binding **binds an HTML Form input element to the property of an object**, so that user can update object properties through GUI (which means the property is editable)

    How to do that using raw JavaScript and DOM? Which JS event to capture?...
  - Templates: HTML view is extended to contain instructions on how the model should be projected into the view (The HTML templates are parsed into the DOM), e.g., special markups/tags, conditional rendering, conditional styling, render two-way bound data…

Angular key features (cont'd)

- Key features
  - MVC: separation of concerns
  - Dependence injection: similar to library import/ reuse existing services/business objects

Angular framework architecture



Angular is an OO framework

You write Angular applications by:

- composing **HTML templates/GUI objects/HTML element objects** with Angularized markup,
- writing **components** to manage those templates,
- adding application logic in **services (reusable)**, like business objects (order, shopping cart…)
- and boxing components and services in **modules**.

Install and launch Angular

- First install node.js, which contains the `npm` (node package manager) application
  - https://nodejs.org/en/download/

  Note that the demo code is based on node-v8.9.4 https://nodejs.org/dist/v8.9.4/

- Then install Angular CLI using in a terminal/command prompt:

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

- To create a new application:

```
ng new my-angular-app
```

- To serve/launch the application:

```
cd my-angular-app
ng serve --open
```

Homepage:
http://localhost:4200/

The root module

The root module

- The root module is the entry point of an app: the app is launched by bootstrapping the root module
- Structure
  - Import required modules from packages
  - Use the `@NgModule` directive to describe the module
  - Export module classes for public reuse

A set of Angular modules work together to build a system:

- *A module provides services to other modules*
- *A module uses services provided by other modules*

Root module example (app.module.ts)

In “src/app” folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-
browser';

@NgModule({
  imports: [ BrowserModule ],
  providers: [ ],
  declarations: [ AppComponent ],
  exports: [ AppComponent ],
  bootstrap: [ AppComponent ]
})

export class AppModule { }
```

Describe the root module

```
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent],
  exports: [ AppComponent ],
  providers: [ ],
  bootstrap: [ AppComponent ] })
```

Add This line to the source code…

- imports - other modules whose exported classes are *needed by component templates* declared in this module.
- declarations - the view objects that belong to this module. Component is a key Angular view object
- exports - the subset of declarations that should be visible and usable in the component templates of other modules.

Describe the root module

```
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent],
  exports: [ AppComponent ],
  providers: [ ],
  bootstrap: [ AppComponent ] })
```

- providers - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app. *(note: we export **components** (view objects), while we provide **services** (reusable functions))*
- bootstrap - the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

Components

Components

- Component = View objects / view controller that defines directives to control how HTML templates are rendered
  - HTML templates are *dynamic*: When Angular renders them, it transforms the DOM according to the instructions given by directives in components

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

@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']})

export class AppComponent {
title = 'app works!';
}
```

This component applies to the <app-root> tag in index.html

Associated html and css files

A component is a class, it has variables/properties

This class variable later on can be used in the template

Bind class variable to HTML element content

- It's called interpolation binding in Angular
- Syntax: double curly braces

app.component.ts:

```
export class
AppComponent {
title = 'app works!';
}
```

app.component.html:

```
<h1>{{title}}</h1>
```

Interpolation binding: changes in one place also apply to the other place…

Creating components

- If we need a new GUI object, such as HTML list, table, canvas, we create a new component
- To create a new component, type in a terminal:

```
ng generate component heroes
```

"heroes" is the Component name

- After that, a new component will appear in app.module.ts (the root module)'s "`declarations`"
- And four new files are created:

```
create src\app\heroes\heroes.component.css
create src\app\heroes\heroes.component.html
create src\app\heroes\heroes.component.spec.ts
create src\app\heroes\heroes.component.ts
```

heroes.component.ts

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

@Component({
selector: 'app-heroes',
templateUrl: './heroes.component.html',
styleUrls: ['./heroes.component.css'] })

export class HeroesComponent implements OnInit {
constructor() { }
ngOnInit() { }
}
```

Note that we haven't put this component onto the GUI yet; it can be put inside any component, such as app.component.html

- The `ngOnInit` is a lifecycle hook Angular calls: where you put initialization logic (e.g., reading data)
- Always export the component class for sharing

Now we can add component logic…

- We can import pure classes in our OO design (such as Hero) in the component
  - *Note: while components are for managing templates/views, pure classes are concepts in the business domain that we identified through requirements analysis*

heroes.component.ts

```
…
import { Hero } from '../hero';

@Component(…)

export class HeroesComponent implements OnInit {
hero: Hero = { id: 1, name: 'Windstorm' };
…
}
```

JSON object…

Create a new `Hero` object called `hero`. We can give variable type now!

hero.ts: in the ‘app’ folder

```
export class Hero {
  id: number;
  name: string;
}
```

Show the hero object in the templates

- heroes.component.html

```
<h2>{{ hero.name }} Details</h2>

<div><span>
id: </span>{{hero.id}}
</div>

<div><span>name:
</span>{{hero.name}}
</div>
```

Interpolation binding…

- app.component.html: add the following tag to display the heroes component

```
<app-heroes></app-heroes>
```

Two-way binding

Two-way binding

- Two-way binding binds an HTML Form input element to the property of an object, so that user can update object properties through GUI (which means the property is editable)
- heroes.component.html

```
<div> <label>name:
<input
[(ngModel)]="hero.name"
placeholder="name"> </label>
</div>
```

Hint to user…

- We also need to import the `FormModule` in the root module (app.module.ts) …

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

@NgModule({…imports: [BrowserModule,FormsModule]
,…})
```

Display a list using *'ngFor'* component directive

Display a list of heroes from data

- mock-heroes.ts:

```
import { Hero } from './hero';

export const HEROES: Hero[] = [
  { id: 11, name: 'Mr. Nice' },
  { id: 12, name: 'Narco' },
  { id: 13, name: 'Bombasto' },
  { id: 14, name: 'Celeritas' },
  { id: 15, name: 'Magneta' },
  { id: 16, name: 'RubberMan' },
  { id: 17, name: 'Dynama' },
  { id: 18, name: 'Dr IQ' },
  { id: 19, name: 'Magma' },
  { id: 20, name: 'Tornado' }
];
```

Then in hero.component.ts, Import this data file and define a new class var:

```
import { HEROES }
from '../mock-
heroes';
...
heroes = HEROES;
```

Display a list of heroes from data

- hero.component.ts

  Import this data file and define a new class var:

```
import { HEROES } from '../mock-heroes';
...
heroes = HEROES;
```

Class var 'heroes'

- hero.component.html
  - Use "ngFor" loop to display the heroes list:

```
<ul>
<li *ngFor="let hero of heroes; let i = index">
{{i}}. {{hero.name}}  </li>
</ul>
```

Event handler

Add an event handler

- heroes.component.html

```
<li *ngFor="let hero of heroes"
(click)="onSelect(hero)">
<span class="badge">{{hero.id}}</span>
{{hero.name}}
</li>
```

- heroes.component.ts

```
selectedHero: Hero;
…
onSelect(hero: Hero): void {
this.selectedHero = hero; }
```

Conditional styling

Conditional styling

- How can we highlight the selected row?
- Use class binding to add CSS class conditionally to an element

heroes.component.html:

```
<li *ngFor="let hero of heroes"
(click)="onSelect(hero)"
[class.selected]="hero === selectedHero">
```

heroes.component.css:

```
.selected {
background-color: #CFD8DC
!important;
color: white;
}
```

Adding a new hero-detail component

The hero detail component

- Used to display the details of a hero

```
HeroDetailsComponent
```

- In terminal type:

```
ng generate component hero-detail
```

- Again, the root module will be updated and four new files are created:

```
create src\app\hero-detail\hero-detail.component.css
create src\app\hero-detail\hero-detail.component.html
create src\app\hero-detail\hero-detail.component.spec.ts
create src\app\hero-detail\hero-detail.component.ts
```

hero-detail.component.html

Display only when selectedHero is not undefined…

Get the variable from the managing component

```
<div *ngIf="selectedHero">

<h2>{{ selectedHero.name}} Details</h2>

<div><span>id:
</span>{{selectedHero.id}}</div>

<div> <label>name: <input
[(ngModel)]="selectedHero.name"
placeholder="name"> </label> </div>

</div>
```

Two-way binding / editable field

hero-detail.component.ts

- As the template needs the property `selectedHero`, it must be defined in the component:
- We also need to import the class definition from hero.ts, and import `Input` package from angular core

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

export class HeroDetailComponent implements OnInit {
@Input() selectedHero: Hero;
...
```

- The `selectedHero` property is an input property, it comes from html template

Bind external value to input properties

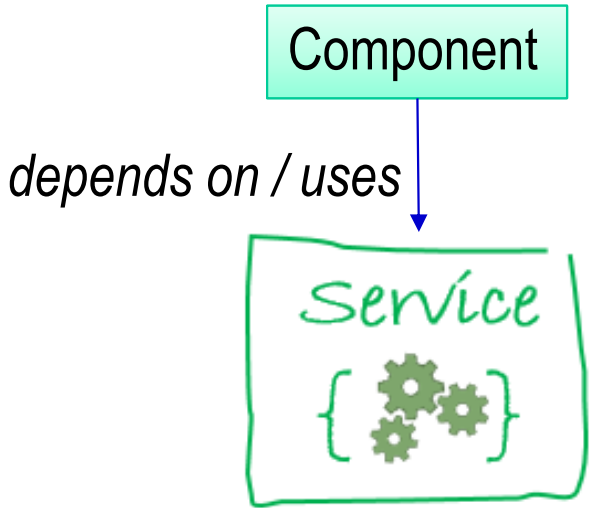
- Update heroes.component.html by appending the following code after `<li *ngFor>` element:

```
<app-hero-detail
[selectedHero]="selectedHero"></app-hero-
detail>
```

Services (optional content)

Services

- A service is a singletons class that provides reusable functionality
- From operating systems / framework middleware level, we have:
  - Data service
  - Logging service
  - Message service
  - …



- We also get external services over the web (e.g., RESTful web services / Web APIs)
- Components are consumers of services

RESTful Web services

- REST: REpresentational State Transfer
- An architectural style
- Organize a distributed application into URI- addressable resources
- Use only the standard HTTP messages -- GET, PUT, POST and DELETE -- to provide the full capabilities of that application

Data service for the `heroes` component

- In terminal, create a service called `hero`

```
ng generate service hero
```

- Note that the service class created is called `HeroService`

- Next we need to provide this service in the root module (app.module.ts) so that all the components can use it

```
import { HeroService } from './hero.service';
…
 providers: [HeroService],
```

hero.service.ts

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

 @Injectable() export class HeroService {
constructor() { }

}
```

- @Injectable means this service can also use (be injected with) other services
- We will see soon that we can inject services to a class in this class's constructor…

The *In-memory Web API* module

- To facilitate development/testing, we can use the Angular In-memory Web API module to simulate remote web services / APIs
- In terminal, type:

```
npm install angular-in-memory-web-api --save
```

- Create a simulated in-memory database called InMemoryDataService (in-memory-data.service.ts)

Alternatively we can also use a RESTful API DB such as MongoDB

in-memory-data.service.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';

export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const heroes = [
      { id: 11, name: 'Mr. Nice' },
      { id: 12, name: 'Narco' },
      { id: 13, name: 'Bombasto' },
      { id: 14, name: 'Celeritas' },
      { id: 15, name: 'Magneta' },
      { id: 16, name: 'RubberMan' },
      { id: 17, name: 'Dynama' },
      { id: 18, name: 'Dr IQ' },
      { id: 19, name: 'Magma' },
      { id: 20, name: 'Tornado' }
    ];
    return {heroes};
  }
}
```

Import in the root module

```
import { HttpClientModule } from
'@angular/common/http';
import { HttpClientInMemoryWebApiModule } from
'angular-in-memory-web-api';
import { InMemoryDataService } from './in-memory-
data.service';
```

- @NgModule({ imports: [

```
HttpClientModule,
// The HttpClientInMemoryWebApiModule
module intercepts HTTP requests // and
returns simulated server responses. //
Remove it when a real server is ready to
receive requests.
HttpClientInMemoryWebApiModule.forRoot(
InMemoryDataService, { dataEncapsulation:
false } )
```

The `forRoot()` configuration method takes an `InMemoryDataService` class that populate the in-memory database.

Update the data service hero.service.ts

- Now that the simulated in-memory DB is ready, we can access it using the data service class: `hero.service`
- We first import HTTP symbols

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

- Inject HttpClient into the constructor in a private property called http.

```
constructor( private http: HttpClient)
```

- Define the heroesUrl with the address of the heroes resource on the server.

```
private heroesUrl = 'api/heroes'; // URL to web api
```

Update the data service hero.service.ts

- Next we create a method `getHeroes` to fetch data from the server (or url)

```
/** GET heroes from the server */
getHeroes (): Observable<Hero[]> { return
this.http.get<Hero[]>(this.heroesUrl) }
```

- Note that the `Observable` interface is from the RxJS library (ReactiveX library for JavaScript), which is used to asynchronously invoke a callback function registered/subscribed to the change of the data

So actually it is till the AJAX concepts we have discussed in previous lectures…

Update the data service hero.service.ts

- We also need to import Hero class definition and the Observable class:

```
import { Hero } from './hero';
import { Observable } from 'rxjs/Observable';
```

heroes.component.ts uses the data service

- Now we can use the simulated web data service to asynchronously fetch the data
- First we inject HeroService into the component:

```
constructor(private heroService: HeroService) { }
```

- Next, we subscribe/register to the data service, so that whenever the data is ready, it is also updated in the component (which finally update the webpage)

```
ngOnInit() {
    this.getHeroes();
}

  getHeroes(): void {
  this.heroService.getHeroes()
  .subscribe(heroes => this.heroes = heroes);
}
```

Add a hero

- To do so, we need to:
  - Create the add method in the data service
  - Create the add event handler in the heroes component
  - Add a new HTML division on the heroes template for getting user input

hero.service.ts – add hero code

- Use HTTP Post to add data to the RESTful DB

```
const httpOptions = { headers: new
HttpHeaders({ 'Content-Type':
'application/json' }) };

//////// Save methods //////////

/** POST: add a new hero to the server */

addHero (hero: Hero): Observable<Hero> {
return this.http.post<Hero>(this.heroesUrl,
hero, httpOptions);
}
```

heroes.component.ts – add hero code

- Defining the event handler

```
add(name: string): void {

name = name.trim();
if (!name) { return; }

this.heroService.addHero({ name } as Hero)
.subscribe(hero => {
this.heroes.push(hero); });

}
```

After ‘add hero’ is successful, also push the data to the local heroes list…

heroes.component.html – add hero code

- Add this division above the heroes list…

```
<div>
<label>Hero name: <input #heroName />
</label>

<!-- (click) passes input value to add()
and then clears the input --> <button
(click)="add(heroName.value);
heroName.value=''"> add </button>

</div>
```

Delete a hero

- To do so, we need to:
  - ☐ Create the delete method in the data service
  - ☐ Create the delete event handler in the heroes component
  - ☐ Add a new HTML delete button to all heroes list items

hero.service.ts – delete hero code

- HTTP DELETE method is used

```
/** DELETE: delete the hero from the server
*/ deleteHero (hero: Hero | number):
Observable<Hero> {

const id = typeof hero === 'number' ? hero :
hero.id;

const url = `${this.heroesUrl}/${id}`;

return this.http.delete<Hero>(url,
httpOptions);

}
```

heroes.component.ts – delete hero code

- Update the local heroes list then asynchronously request the data service to delete the selected hero

```
delete(hero: Hero): void
{

this.heroes = this.heroes.filter(h => h
!== hero);

this.heroService.deleteHero(hero).subscr
ibe();

}
```

heroes.component.html – delete hero code

- The HTML code for the button…
- Note that the selected hero has been bound to the input parameter `hero`

```
<ul class="heroes">
<li *ngFor="let hero of heroes">
…

<button class="delete" title="delete hero"
(click)="delete(hero)">x</button>

</li>

</ul>
```

References

- angular.io/tutorial
- https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282
- http://thelillysblog.com/2016/11/02/MEAN-stack-with-Angular2/