UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 21

ANGULAR: PART II

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace

PREVIOUSLY, ON WEB TECHNOLOGIES

- In Lect. 19, we started learning about Angular
- We know what **Components** are, how to use the built-in **Router**, and how to manage @Input/@Output from Components
- Today, we'll see some other important features of the Angular Framework. Notably:
  - Forms
  - Services and Dependency Injection
  - More about the Angular Router (securing routes)
  - HTTP Client
  - Signals

ANGULAR FORMS

- Forms are in many cases the preferred way of getting user input
- In Angular, forms can be handled in two ways:
  - **Template-driven forms**: associations between Component data and form are defined implicitly, via template directives
  - **Reactive forms**: associations between Component data and form are defined explicitly in the component

ANGULAR FORMS

Let’s add a Login form to our app. We’ll start by creating a component

```
@luigi ➜ D/O/T/W/2/e/2/angular-app $ ng generate component login
```

We associate the component with a new **Route** on the **/login** path

```
{ // new Route in app.routes.ts
   path: "login",
   title: "Login",
   component: LoginComponent
}
```

Last, we add a link in the **app.component.ts** template

```
// in the template for the AppComponent, before <router-outlet/>
<a routerLink="/login" routerLinkActive="active">Login</a>
```

ANGULAR FORMS: TEMPLATE–DRIVEN

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

@Component({
  selector: 'app-login’,
  standalone: true,
  imports: [FormsModule],
  template: './login.component.html',
  styleUrl: './login.component.scss'
})
export class LoginComponent {
  user: string = "";
  pass: string = "";
}
```

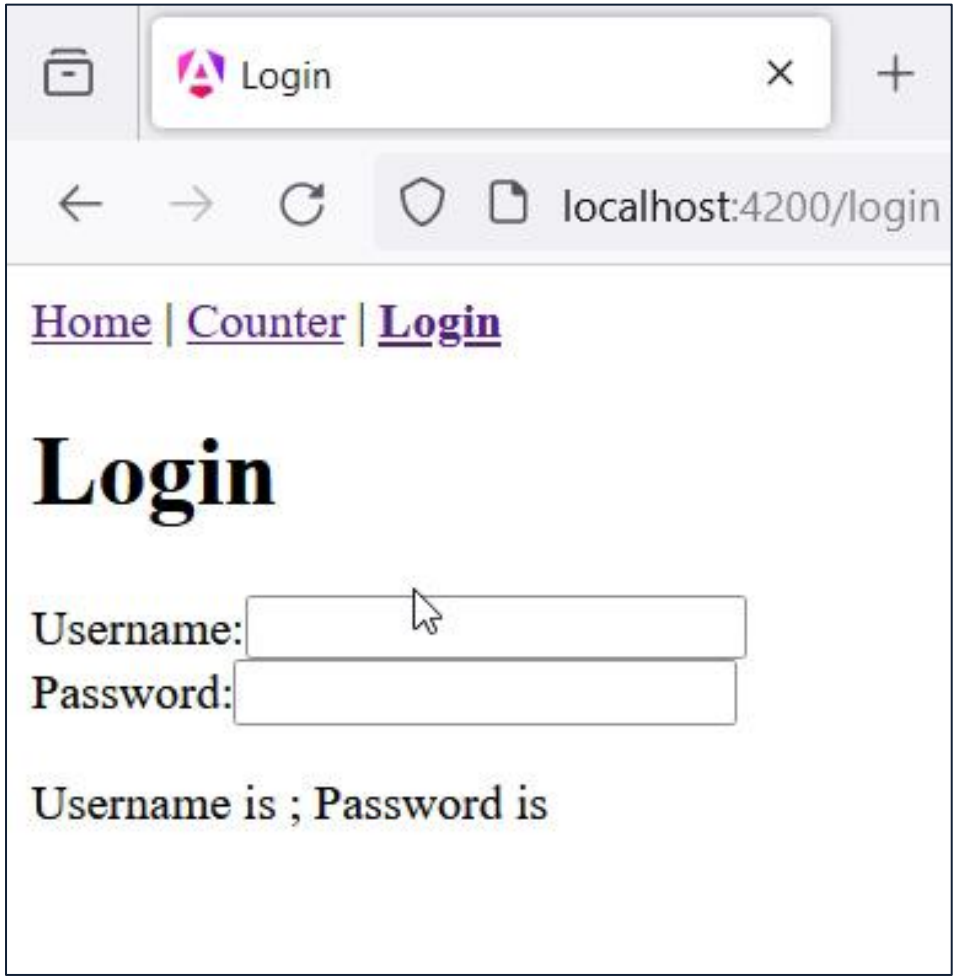
```
<h1>Login</h1>
<form>
  <label for="usr">Username:</label>
  <input id="usr" name="usr"
         [(ngModel)]="user"/>
  <label for="pwd">Password:</label>
  <input id="pwd" name="pwd"
         type="password"
         [(ngModel)]="pass"/>
</form>
<p>
  Username is <strong>{{user}}</strong>;
  Password is <strong>{{pass}}</strong>
</p>
```

ANGULAR FORMS: TEMPLATE–DRIVEN

- The `[(ngModel)]="pass"` directive is used to specify that the value of a form field is bound to a property (`pass`, in the example) of the component.
- This `[()]` notation is also known as «**banana in a box**» and represents a **two-way binding**: property binding and event binding

ANGULAR FORMS: TEMPLATE–DRIVEN

- As we type in the input, data in the component is updated automatically, and the template is updated as well
- If some change occurred to a property in the component, the corresponding input field would automatically update (**two-way binding**)
- For example, try to initialize the **user** property to a different value!



ANGULAR FORMS: REACTIVE APPROACH

```ts
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ReactiveFormsModule],
  template: './login.component.html',
  styleUrl: './login.component.scss'
})
export class LoginComponent {
  loginForm = new FormGroup({
    //argument is initial value
    user: new FormControl(''),
    pass: new FormControl('')
  })
}
```

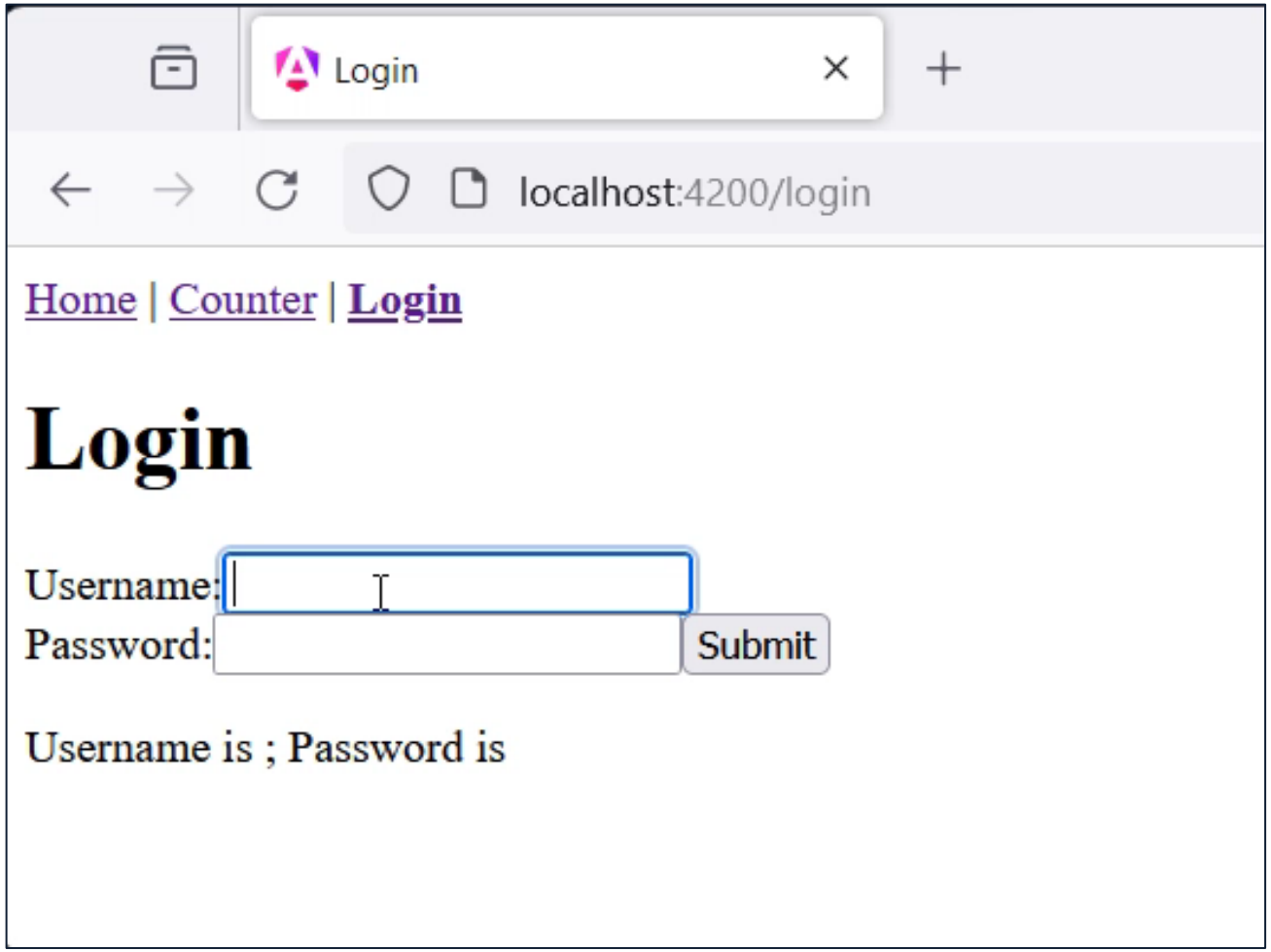
```html
<h1>Login</h1>
<form [formGroup]="loginForm">
  <label for="usr">Username:</label>
  <input id="usr" formControlName="user"/>
  <label for="pwd">Password:</label>
  <input id="pwd" formControlName="pass"
         type="password"/>
  <button type="submit">Submit</button>
</form>
<p>
Username is
<strong>{{loginForm.value.user}}</strong>;
Password is
<strong>{{loginForm.value.pass}}</strong>
</p>
```

ANGULAR FORMS: SUBMISSIONS

```
export class LoginComponent {
  loginForm = new FormGroup({
    user: new FormControl(''), pass: new FormControl('')
  })

  onSubmit(){
    alert(`Username: ${this.loginForm.value.user}
           Password: ${this.loginForm.value.pass}`);
  }
}
```

```
<form [formGroup]="loginForm" (ngSubmit)="onSubmit()">
  <input id="usr" formControlName="user"/>
  <input id="pwd" formControlName="pass" type="password"/>
  <button type="submit">Submit</button>
</form>
<p>
  Username is <strong>{{loginForm.value.user}}</strong>;
  Password is <strong>{{loginForm.value.pass}}</strong>
</p>
```

ANGULAR FORMS: SUBMISSION



SERVICES AND DEPENDENCY INJECTION

INTRODUCING SERVICES

- We know that classes in our software should do **one thing** and **do it right**. This increases modularity and reusability.
- Components in Angular should be only responsible for enabling the user experience: template, styles, and logic that mediates between the view and the application logic

ANGULAR: SERVICES

- Services are classes that provide functionality that can be used in multiple parts of an application
- Services are **not** associated directly with user experience
  - No template, no styles. Just logic, that can be invoked wheen needed
- Services are great for tasks such as fetching data from a REST API, logging, validating data, etc…
  - These kind of tasks are likely to be required in different parts of the application

DEPENDENCY INJECTION (DI)

- Angular leverages the DI pattern to make it easy for application code to obtain instances of its dependecies (e.g.: Services it needs)
- The goal of DI is to separate the concerns of (1) constructing dependencies and (2) using them, improving decoupling
- In the Angular DI system two roles exist:
  - **dependency consumer** (i.e.: the code we write)
  - **dependency provider** (generally managed by Angular)

ANGULAR: CREATING SERVICES

```
@luigi ➜ D/O/T/W/2/e/2/angular-app $ ng generate service calculator

CREATE src/app/calculator.service.spec.ts (377 bytes)
CREATE src/app/calculator.service.ts (139 bytes)
```

- The **@Injectable** decorator is used to mark the CalculatorService as something that can be injected into a component as a dependency

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

@Injectable({
  providedIn: 'root' //avilable everywhere in the app (root injector)
})
export class CalculatorService {
  sum(x: number, y: number): number { return x+y; }
}
```

ANGULAR: INSTANTIATING SERVICES

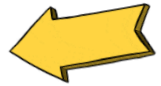
Dependencies can be instantiated using the **inject()** method

```
import { Component, inject } from '@angular/core';
import { CalculatorService } from '../calculator.service';

@Component({
  selector: 'app-home-page',
  standalone: true,
  imports: [],
  template: "<p>The sum is <strong>{{sum}}</strong></p>",
  styleUrl: './home-page.component.scss'
})
export class HomePageComponent {
  private calculatorService = inject(CalculatorService);
  sum = this.calculatorService.sum(1234, 4321);
}
```

ANGULAR: INSTANTIATING SERVICES

Dependencies can also be injected by declaring them in class constructors

```
import { Component } from '@angular/core';
import { CalculatorService } from '../calculator.service';

@Component({
  selector: 'app-home-page',
  standalone: true,
  imports: [],
  template: "<p>The sum is <strong>{{sum}}</strong></p>",
  styleUrl: './home-page.component.scss'
})
export class HomePageComponent {
  constructor(private calculatorService: CalculatorService){}
  sum = this.calculatorService.sum(1234, 4321);
}
```



SECURING ROUTES

SECURING ROUTES

- A common task when handling Routing is to ensure that only authorized users access certain routes
- With the Angular Router, this can be done using **Route Guards**
- **Route Guards** are functions that are invoked by Angular to determine whether the user is allowed to perform certain operations (e.g.: canActivate, canMatch, canLoad, canDeactivate,…) on a Route
- Guards are specified as properties of **Route** objects
- Multiple guards can be specified for a Route. Angular executes all guards in parallel, and the action can be performed only if all guards return **true**.

CREATING A GUARD

```
@luigi → D/O/T/W/2/e/2/angular-app $ ng generate guard authorization
? Which type of guard would you like to create? CanActivate
CREATE src/app/authorization.guard.spec.ts (497 bytes)
CREATE src/app/authorization.guard.ts (137 bytes)
```

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from './auth.service';

export const authorizationGuard: CanActivateFn = (route, state) => {
  return true;
};
```

CREATING A GUARD

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from './auth.service';

export const authorizationGuard: CanActivateFn = (route, state) => {
  const router = inject(Router);
  const authService = inject(AuthService);
  if(authService.isUserLoggedIn()){
    return true; //user is authorized
  } else {
    router.navigateByUrl("/login"); //redirect to login
    return false; //user is not authorized
  }
};
```

```
{ //in app.routes.ts
  path: "counter",
  title: "Counter",
  component: CounterPageComponent,
  canActivate: [authorizationGuard]
}
```

USING GUARDS

- Guard functions can return a boolean value or an **UrlTree** object
  - `true` means that the guard is satisfied an navigation may proceed
  - `false` when the guard is not satisfied and the action may not be performed
  - an UrlTree object means that Router should redirect to that URL
- When multiple guards are set, you should always return an UrlTree to redirect, and not use the Router.navigate() methods directly like in the example shown in the previous slide
  - When multiple guards are executed in parallel, if two of them redirect, you may have race conditions!

HTTPCLIENT AND INTERCEPTORS

HTTPCLIENT

- Angular provides an API to perform HTTP requests
- It's implemented in the **HttpClient** service class
- To use HttpClient, we need to configure it using **dependency injection**

```
//in app.config.ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(), //add the providers using this dedicated helper function
    provideRouter(routes)
  ]
};
```

USING HTTPCLIENT

- HttpClient can then be instantiated as a dependency of our services and components (e.g.: using **inject()** or by declaring it in a class constructor)

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

@Injectable({
  providedIn: 'root'
})
export class RestBackendService {
  constructor(private http: HttpClient) {}
  //This service can now make HTTP requests via `this.http`.
}
```

MAKING HTTP REQUESTS

```
@Component({
  selector: 'app-cat-facts',
  standalone: true,
  template: `<h1>Cat Facts</h1>
    <button (click)="loadCatFact()">Click here to load a cat fact</button>
    <p>{{catFact}}</p>`
})
export class CatFactsComponent {
  constructor(private http: HttpClient){}
  catFact: string = "";
  loadCatFact(){
    let url = "https://cat-fact.herokuapp.com/facts/random";
    this.http.get<{text: string}>(url).subscribe(data => {
      this.catFact = data.text;
    });
  }
}
```

USING HTTPCLIENT: OBSERVABLES

- HttpClient provides methods corresponding to the different HTTP verbs used to make requests
- Each method returns an **Observable** from the RxJS library
  - Reactive Extensions Library for JavaScript
- Angular uses Observables to manage many async operations
- In some ways, Observable are similar to Promises. In a nutshell:
  - Promises can resolve (or reject) only once. In other words, they asynchronously emit a single value (or error)
  - Observables can asynchronously emit **multiple values** over time

PROMISES: REDUX

```
let promise: Promise<number> = new Promise((resolve, reject) => {
  let num = Math.random();
  setTimeout( () => {
    if(num < 0.5){
      resolve(num);
    } else {
      reject(new Error(`You were unlucky! ${num}`));
    }
  }, 2000); //run the callback in 2 seconds
});

promise.then( (data) => {
  console.log(`Promise resolved returning ${data}`);
}).catch( err => {
  console.log(`Promise rejected with ${err}`);
});
```

WORKING WITH OBSERVABLES

- Just as with Promises, you need to provide callbacks to execute when some events occur.
- This is done by **subscribing** to the Observable

```
observable.subscribe({//provide a Subscriber object
  next(value){//executed when the a value is emitted by the Observable
    console.log(`Observable emitted value ${value}`);
  },
  error(err){ //executed when an error is emitted by the Observable
    console.log(`Observable emitted an Error: ${err}`);
  },
  complete(){ //executed when the Observable is done (will not emit any more values)
    console.log("Observable finished emitting values");
  }
});
```

CREATING AN OBSERVABLE

- The argument passed to the Observable constructor is the function to execute upon subscription

```
import { Observable } from "rxjs";

const observable = new Observable<number>( (subscriber) => {
  setTimeout( () => {
    let num = Math.random();
    if(num < 0.5){
      subscriber.next(num);
    } else {
      subscriber.error(new Error(`You were unlucky! ${num}`));
    }
    subscriber.complete();
  }, 2000)
});
```

CREATING AN OBSERVABLE

```
import { Observable } from "rxjs";

const observable = new Observable<number>( (subscriber) => { /*...*/ });

observable.subscribe({
  next(value){ console.log(`Observer emitted value ${value}`  )},
  error(err) { console.log(`Observer emitted an Error: ${err}`)},
  complete() { console.log("Observer finished emitting values")}
});
```

```
@luigi → D/O/T/W/2/e/2/a/src $ tsc .\observables_example.ts
@luigi → D/O/T/W/2/e/2/a/src $ node .\observables_example.js
Observer emitted value 0.3339789464735569 7
Observer finished emitting values
@luigi → D/O/T/W/2/e/2/a/src $ node .\observables_example.js
Observer emitted an Error: You were unlucky! 0.620240538767127
@luigi → D/O/T/W/2/e/2/a/src $
```

Notice that errors are considered as completions! **complete()** will not be called when errors occur!

OBSERVABLES: EMITTING MULTIPLE VALUES

```
import { Observable, Subscriber } from "rxjs";

const observable = new Observable<number>( subscriber => {
  helper(subscriber, 1);
});

function helper(subscriber: Subscriber<number>, level: number){
  setTimeout( () => {
    let num = Math.random();
    if(level > 3){ subscriber.complete(); return; }
    if(num < 0.8) {
      subscriber.next(num);
      helper(subscriber, level+1); //recursion
    } else { subscriber.error(new Error(`You were unlucky! ${num}`)); }
  }, 1000);
}
```

OBSERVABLES: EMITTING MULTIPLE VALUES

```
const observable = new Observable<number>( (subscriber) => { /*...*/ });

observable.subscribe({
  next(value){ console.log(`Observer emitted value ${value}`  )},
  error(err) { console.log(`Observer emitted an Error: ${err}`)},
  complete() { console.log("Observer finished emitting values")}
});
```

```
@luigi → D/O/T/W/2/e/2/a/src $ node .\observables_example.js
Observer emitted value 0.6982903488018062
Observer emitted value 0.5713928193844571
Observer emitted value 0.005029574874092724
Observer finished emitting values
@luigi → D/O/T/W/2/e/2/a/src $ node .\observables_example.js
Observer emitted value 0.09009756551320991
Observer emitted an Error: Error: You were unlucky! 0.8901570954122466
@luigi → D/O/T/W/2/e/2/a/src $
```

SETTING UP HTTPCLIENT

- We can customize many **HttpClient** features (see setup guide) when we configure its DI providers:
  - We can use **withFetch()** to use the Fetch API instead of **XMLHttpRequest**
  - We can configure **Interceptors**

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(
      withFetch(), //use the Fetch API instead of XMLHttpRequests
      withInterceptors([authInterceptor])
    ),
    provideRouter(routes)]
};
```

INTERCEPTORS

- Interceptors are a form of **middleware** supported by HttpClient
- Same concept as middlewares in Express, but apply to outgoing requests!
- Ultimately, interceptors are functions that take as input
  - The current outgoing **HttpRequest**
  - A **next** function, representing the next step in the interceptor chain

```
function log(req: HttpRequest, next: HttpHandlerFn): Observable {
  console.log(req.url);
  return next(req);
}
```

INTERCEPTORS: USE CASES

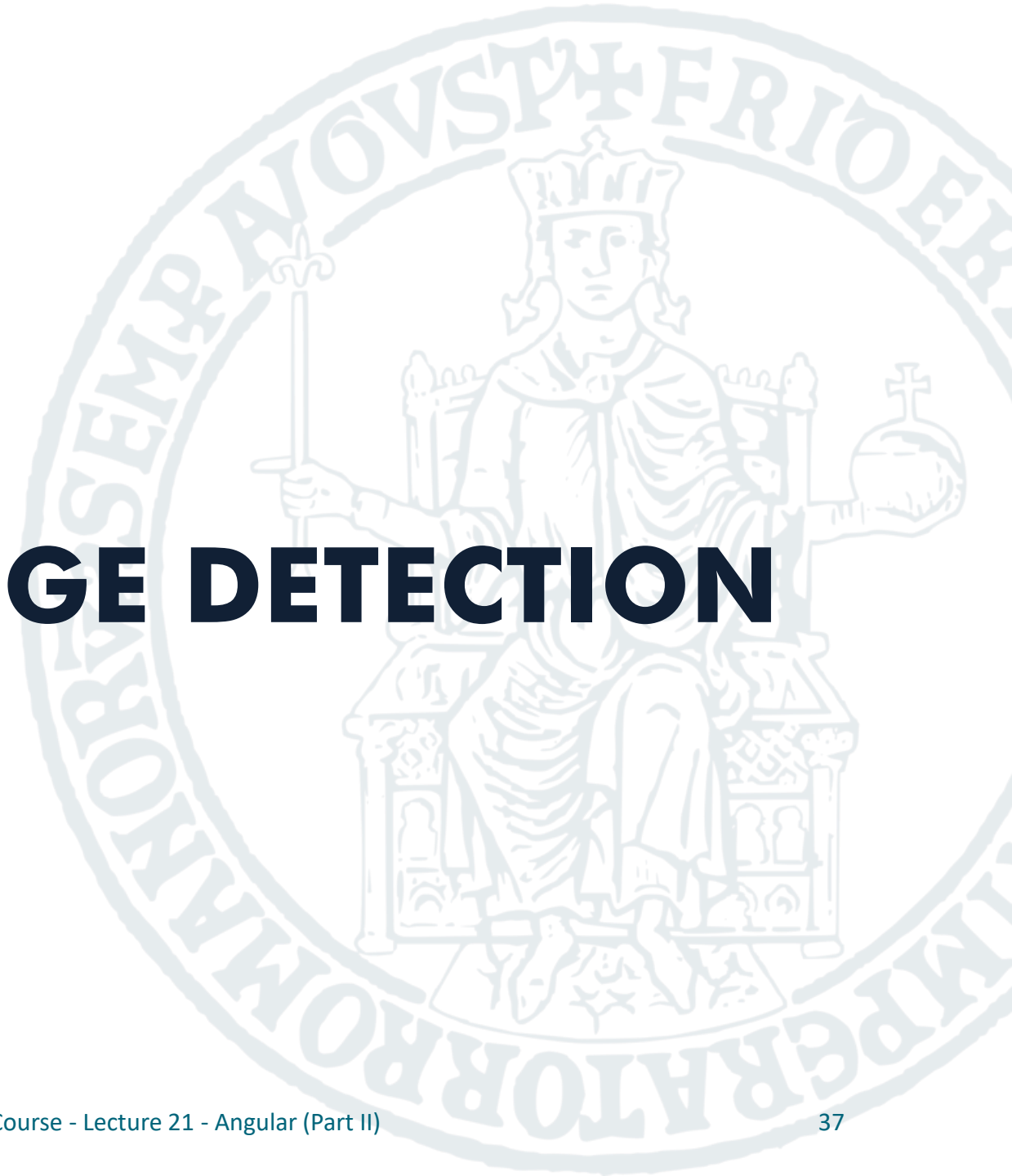
Interceptors can be used to

- Add authentication headers to requests (e.g.: JWT tokens!)
- Cache responses for a period of time
- Log outgoing requests

- Interceptors to use are declared when configuring **HttpClient**

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(
      withInterceptors([loggingInterceptor, authInterceptor])
    )
  ]
}
```

ANGULAR: CHANGE DETECTION



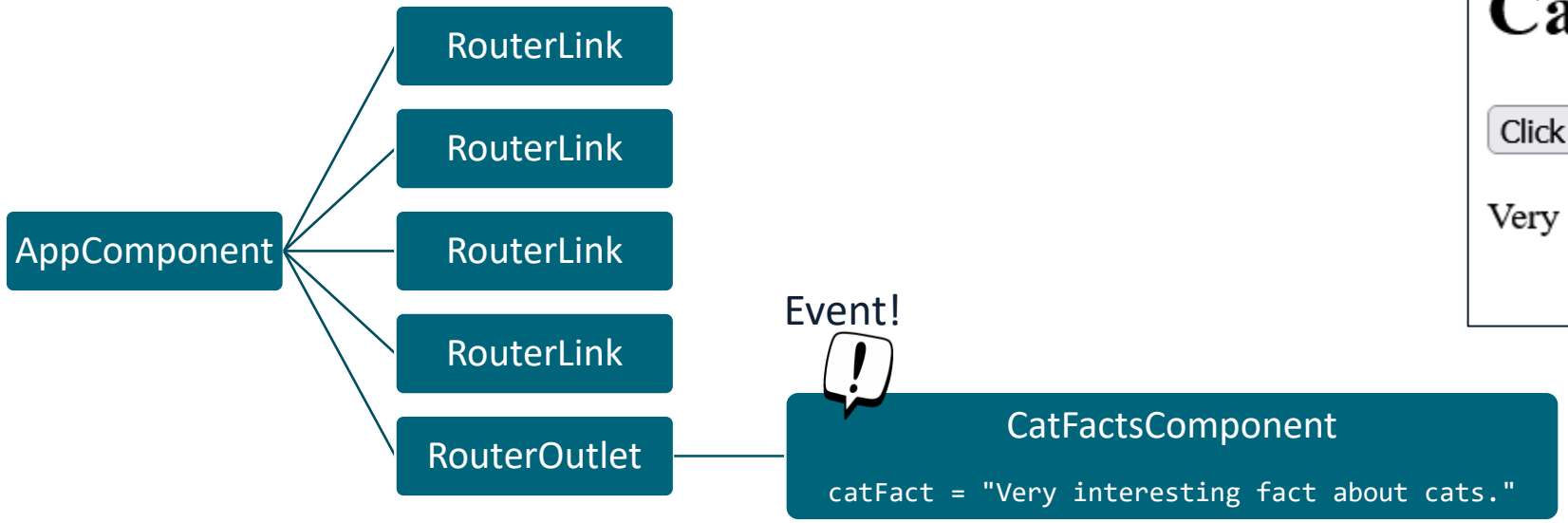
ANGULAR: CHANGE DETECTION

- When a component property (i.e.: its **state**) changes, its template is automatically updated by Angular
- Updating the templates only when necessary and doing so efficiently (e.g.: updating only the parts that need to be updated) is key to good SPA performance
- How does Angular know when it needs to update the rendered view?

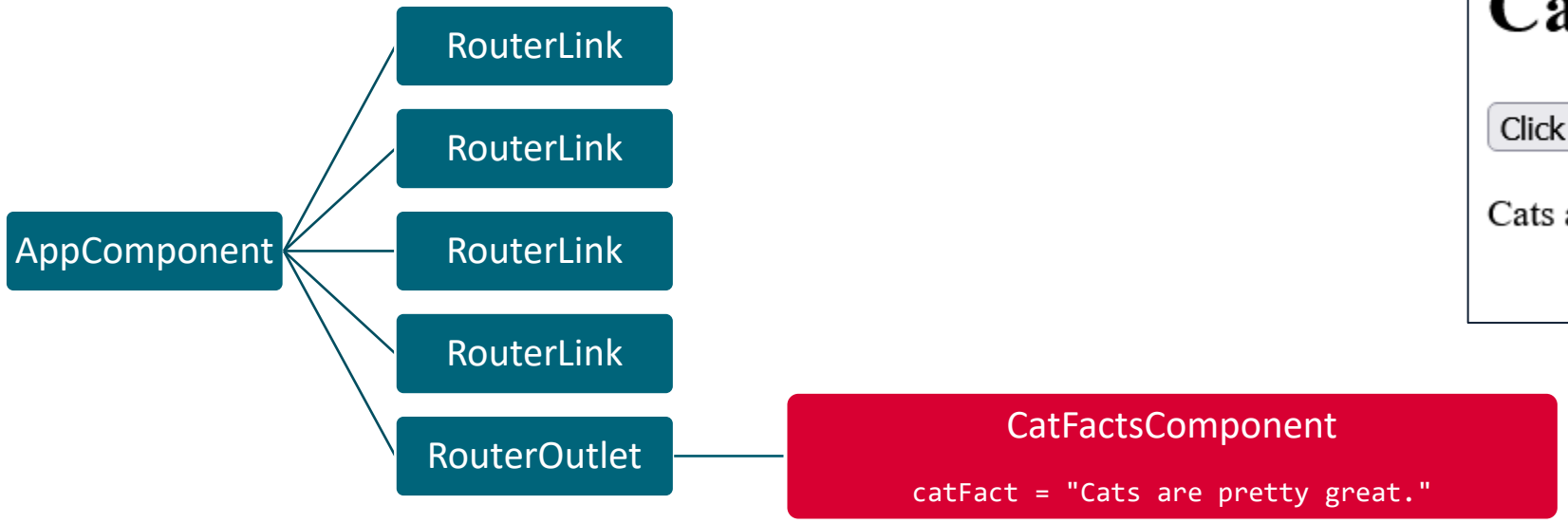
ANGULAR: CHANGE DETECTION

- Basically, when something happens in the application (DOM events, async operations, …) the **change detection** mechanism triggers
- When change detection starts, Angular goes through all the components in the component hierarchy
  - For each component, it checks whether its state changed, and whether the state change affects the view
  - If so, the DOM portion corresponding to the component template is updated
- We won’t delve into the internals of the change detection process
  - Check out the docs if you want the details

ANGULAR: CHANGE DETECTION EXAMPLE



ANGULAR: CHANGE DETECTION EXAMPLE



SIGNALS

ANGULAR SIGNALS

- Signals were introduced in Angular 17
- They granularly track how and where state is used throughout an app, allowing Angular to further optimize rendering updates.
- In a nutshell, you can think of signals as variables that can also notify any interested consumers when their value changes
- They are a way to achieve **reactive programming**
  - A form of declarative programming based on asynchronous event processing
- Using Signals allows Angular to avoid unnecessary change detection checks and updated only the parts of the DOM that changed

CREATING AND UPDATING SIGNALS

- Signals can be **writeable** or **read-only**
- Writeable signals can be created using the **signal()** function
- When creating a writeable signal, we pass an initial value to **signal()**
- Signals provide an API for updating their values
  - **.set()** allows to set a new value
  - **.update()** allows to compute a new value based on the previous one

```
export class CounterSignalComponent {
  count = signal(0); //initial value is zero
  handleClick() { this.count.update(value => value + 1); }
  handleReset() { this.count.set(0); }
}
```

USING SIGNALS: EXAMPLE

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

@Component({
  selector: 'app-counter-signal',
  standalone: true,
  template: `
    <h1>Counter with Signals</h1>
    <button (click)="handleClick()">Click to Increment</button>
    <button (click)="handleReset()">Click to Reset</button>
    <p>{{count()}}</p> <!-- notice the ()! -->
  `
})
export class CounterSignalComponent {
  count = signal(0); //initial value is zero, returns a function to get the value
  handleClick() { this.count.update(value => value + 1); }
  handleReset() { this.count.set(0); }
}
```

COMPUTED SIGNALS

- Computed signals are **read-only** signals that derive their values from those of other signals
- Defined using the **computed()** function and a derivation function
- Angular **automatically** keeps them updated, in a reactive fashion
- Using **.set()** or **.update()** on them will result in an error!

```
export class CounterSignalComponent {
  count = signal(0); //initial value is zero
  double = computed(() => this.count() * 2);
  //rest of the component omitted
}
```

EXAMPLE WITH COMPUTED SIGNALS

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

@Component({
  selector: 'app-counter-signal',
  standalone: true,
  template: `
    <h1>Counter with Signals</h1>
    <button (click)="handleClick()">Click to Increment</button>
    <button (click)="handleReset()">Click to Reset</button>
    <p>Count: {{count()}} - Double: {{double()}}</p> <!-- notice the ()! -->
  `
})
export class CounterSignalComponent {
  count = signal(0); //initial value is zero, returns a function to get the value
  double = computed(() => this.count() * 2);
  handleClick() { this.count.update(value => value + 1); }
  handleReset() { this.count.set(0); }
}
```

SIGNALS: EFFECTS

- Signals notify interested consumers when they change
- An **effect** is a function that runs whenever some signals change
- Effects can be created using the **effect()** function

```
effect(() => {
  console.log(`Count value changed: ${this.count()}`);
})
```

- Effects always run at least once.
- When they run, they keep track of any signal they read.
- Whenever any of these signals change, the effect is executed again

SIGNALS: EFFECTS

- Recall effects run at least once!

```
export class CounterSignalComponent {
  count = signal(0); //initial value is zero
  double = computed(() => this.count() * 2);

  constructor(){
    effect(() => {
      console.log(`Count value changed: ${this.count()}`);
    })
  }

  handleClick() { this.count.update(value => value + 1); }

  handleReset() { this.count.set(0); }
}
```

Console log output
Count value changed: 0
Count value changed: 1
Count value changed: 2
…

REFERENCES

- **Angular Docs**
  https://angular.dev/overview
  Relevant parts: Introduction: Essentials; In-depth Guides: Forms, Dependency Injection, Router, HTTP Client, Signals.