



# **Reactive Forms, Angular Router**

**CS572 – Modern Web Application**

**Maharishi International University**

**Department of Computer Science**

**Associate Professor Asaad Saad**

# Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

# Forms in Angular

A lot of applications are very form-intensive, especially for enterprise development.

Forms can end up being really complex:

- Form inputs are meant to **modify data**, both on the page and the server
- Users cannot be trusted in what they enter, so you need to **validate** values
- The UI needs to clearly **state expectations** and errors
- **Dependent fields** can have complex logic
- We want to be able to **test** our forms, without relying on DOM selectors

Form states such as being: **valid, invalid, pristine, dirty, untouched, touched, disabled, enabled, pending..etc**

# Choosing an approach

Angular offers two approaches to work with forms:

## **Reactive forms** (Data-driven forms)

Provide direct, explicit access to the underlying form's object model. Compared to template-driven forms, they are more **robust**: they're more **scalable**, **reusable**, and **testable**. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

## **Template-driven forms**

Rely on directives in the template to create and manipulate the underlying object model. They **don't scale** as well as reactive forms. If you have very **basic form requirements and logic** that can be managed solely in the template, template-driven forms could be a good fit.

# Angular Form Models

**FormGroup** is an aggregation of the values of its children.

**FormControl** corresponds to a simple UI element, such as an input. It has a **value**, **status**, and a map of **errors**

**FormArray** A collection that has multiple FormControl elements.



Every object includes observables: **statusChanges**, **valueChanges**, **events**

The **Validators** interface allows us to add built-in validation rules.



# Why Form Model?

The form model is a UI-independent way to represent user input consist of simple controls (`FormControl1`) and their combinations (`FormGroup` and `FormArray`), where each control has a value, status, validators, errors, it emits events and it can be disabled.

Having this model has the following advantages:

- Form handling is a complex problem. Splitting it into UI-independent and UI-dependent parts makes them easier to manage. And we can test form handling without rendering UI.
- Having the form model makes reactive forms possible.

# Form Parts

## Form Model

Create the form models: `FormGroup`, `FormControl`, and `FormArray`.

## DOM

Create your DOM form elements `<form/>`, `<input/>`, `<select/>`, `<textarea/>`.

## Form Directives

Connect the above two parts with form directives: `[formGroup]`, `[formControl]`.

# FormBuilder Service

```
form = inject(FormBuilder).nonNullable.group({  
    email: '',  
    password: ''  
})
```

// to pass validators:

```
form = inject(FormBuilder).nonNullable.group({  
    'form_field': ['default value',  
        ValidatorFn | ValidatorFn[],  
        AsyncValidatorFn | AsyncValidatorFn[] ]  
});
```

// or use the AbstractControlOptions as follows:

```
form = inject(FormBuilder).nonNullable.group({  
    'form_field': ['default value', {  
        validators?: ValidatorFn | ValidatorFn[];  
        asyncValidators?: AsyncValidatorFn | AsyncValidatorFn[];  
        updateOn?: "change" | "blur" | "submit"; }]  
});
```

It helps building type-safe and reactive forms. The **nonNullable** does not affect the possibility of having undefined as a value, which should also be checked.



# Form API

```
this.form.value // { email: '', password: '' }
```

```
this.form.valid // true or false
```

```
this.form.patchValue({ email: 'asaad@miu.edu', password: '123456' })
```

```
this.form.controls.email.value // reading the email value
```

```
this.form.controls.email.valid // true or false
```

```
this.form.controls.email.patchValue('theo@miu.edu') // update the email value
```

```
this.form.controls.email.hasError('required') // reading validation errors
```

```
this.form.controls.email.setErrors({required: true}) // invalidate email
```

```
this.form.controls.email.setErrors(null) // set email as valid
```

# Watching For Changes

Both `FormGroup` and `FormControl` have two built-in `EventEmitter` objects that we can use to observe changes.

```
this.form.statusChanges // observable
this.form.valueChanges // observable
this.form.events // observable, combine subscriptions to valueChanges and statusChanges

this.form.controls.email.statusChanges // observable
this.form.controls.email.valueChanges // observable
this.form.controls.email.events // observable
```

Remember to **unsubscribe** when you unmount the component.

# Unified control state change events

The events observable emits four distinct types of events:  
**PristineEvent**, **StatusEvent**, **TouchedEvent**, **ValueChangeEvent**.

```
control.events.subscribe(event => {  
    if(e instanceof StatusEvent) {  
        console.log(e.status)  
    }  
    if(e instanceof ValueChangeEvent) {  
        console.log(e.value)  
    }  
    if(e instanceof PristineEvent) {  
        console.log(e.pristine)  
    }  
    if(e instanceof TouchedEvent) {  
        console.log(e.touched)  
    }  
});
```

# Connecting DOM to Model

```
@Component({
  imports: [ReactiveFormsModule],
  template: `<form [formGroup]="form" (ngSubmit)="onSubmit()">
    <input type="text" [formControl]="form.controls.email" />
    @if(!form.controls.email.valid){ <div>Invalid email</div> }
    <input type="text" [formControl]="form.controls.password" />
    <button type="submit" [disabled]="!form.valid">Submit</button>
  </form>`
})
```

Inspect the code and see how angular adds state change classes to the form elements.

# Display Errors Gracefully

```
@Component({
  imports: [ReactiveFormsModule],
  template: `
    <form [formGroup]="form">
      <input [formControl]="email" />
      @if(email.invalid && (email.dirty || email.touched)){
        <div>
          @if(email.hasError('required')){<div>Email is required</div>}
          @if(email.hasError('email')){<div>Email is not valid</div>}
        </div>
      }
    </form>
  `,
})
export class App {
  form = inject(FormBuilder).nonNullable.group({
    email: ['', [Validators.required, Validators.email]]
  })
  get email() { return this.form.controls.email; }
}
```

# Custom Synchronous Validator

A validator is a function that takes a **AbstractControl** as an input and returns a **Record<string, boolean>** where the key is error name and the value is set to **true** if it fails

```
customValidator(control: AbstractControl): {[s: string]: boolean} | null {  
    return control.value === 'Example'? { exampleError: true } : null  
}
```

```
@if(form_field.hasError('exampleError')){ <div>Example is not valid</div> }
```

# Custom Cross-Validation Sync Validator

```
public form: FormGroup = inject(FormBuilder).nonNullable.group({
  email: '',
  password: '',
  confirm_password: ''
}, { validators: this.match_password })

match_password(controls: AbstractControl) {
  return controls.get('password')?.value === controls.get('confirm_password')?.value
    ? null
    : { mismatchError: true }
}

@if(form.hasError('mismatchError')){ <div>Passwords do not match</div> }
```

# Custom Asynchronous Validator

Asynchronous validator is a function that takes a **FormControl** as its input and returns a **Promise<any>** or an **Observable<any>**

```
asyncValidator(control: AbstractControl): Promise<any> | Observable<any> {  
    // mimic HTTP request  
    return of(null).pipe(delay(1500)) // valid, mimic delay after 1500ms  
}
```

## Notes

- Because async validators run asynchronously, make sure you bind **this** to the component instance.
- While the promise or the observable is being resolved, the status of the form/control will be **PENDING**
- You can delay updating the form validity by changing the **updateOn** property from **change** (default) to **submit** or **blur**. You may also replace the async validator by subscribing to **valueChanges** stream and add a **debounceTime** delay before running your custom validation logic.



# Routing

Routing means splitting the application into different areas usually based on rules that are derived from the current URL in the browser.

Defining routes in our application is useful because we can:

- Separate different areas of the app

- Maintain the state in the app

- Protect areas of the app based on certain rules

# Components of Angular routing

There are three main components that we use to configure routing in Angular:

- Routes** describe a map between URLs and components.

- Router Outlet** is a placeholder where components are mounted and unmounted.

- The routerLink** directive is used instead of href attribute to link to routes so our browser won't refresh when we change routes.

# Angular Router Setup

We can specify that a route takes a parameter by putting a colon in front of the path segment like this `/route/:param`

```
const appRoutes: Routes = [  
  { path: '', redirectTo: 'home', pathMatch: 'full' },  
  { path: 'home', component: Home, title: 'Homepage' },  
  { path: '**', redirectTo : 'home' }  
];
```

Sets up providers necessary to enable Router functionality for the application.

```
bootstrapApplication(App, {  
  providers: [provideRouter(appRoutes)]  
});
```

# Router Outlet <router-outlet/>

When we change routes, we want to keep our outer layout template and only substitute the inner section of the page with the route's component.

In order to describe to Angular **where** in our page we want to render the contents for each route, we use the **router-outlet** directive.

We are going to use our AppComponent as a layout.

# RouterLink [routerLink]

If we tried creating links that refer to the routes directly using pure HTML, it will result to links when clicked they trigger a GET request and cause a **page reload**

```
<a href="/home">Home</a>
```

To solve this problem, we will use the **routerLink** directive:

```
<a [routerLink]="['home']">Home</a>
```

# Example of Layout Component

```
@Component({
  selector: 'AppComponent',
  imports: [RouterOutlet, RouterLink],
  template: `
    <nav>
      <ul>
        <li><a [routerLink]="['home']">Home</a></li>
        <li><a [routerLink]="['products']" [queryParams]="{ page: 1 }">About</a></li>
        <li><a [routerLink]="['contact']">Contact us</a></li>
      </ul>
    </nav>
    <router-outlet />`
})
class App {}
```

Using [routerLink] will instruct Angular to take ownership of the click event and then initiate a route switch to the right place, based on the route definition.

# Reading Params and QueryParams as Inputs

The binding works with routed components for parameters, query parameters, and resolved data.

```
provideRouter(routes, withComponentInputBinding())
```

```
// create a signal within the routed component that matches the name of the  
parameter.
```

# Imperative Routing

You can also navigate to a route imperatively (in your code), you need to inject the **Router** service then you may call **navigate()** like this:

```
#router = inject(Router);
```

```
this.#router.navigate(['home'])
```

```
this.#router.navigate(['users'], { queryParams: { page: 2 } }) // users?page=2
```



# View Transitions API

The View Transitions API enables smooth transitions when changing the DOM. The feature uses the browser's native capabilities for creating animated transitions between routes.

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter, withViewTransitions } from '@angular/router';
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes, withViewTransitions()),
    provideAnimationsAsync()
  ]
};
```

Defer loading of the animations module: **provideAnimationsAsync()** allows you to lazily load the animation module and shaves 60KBs from your initial bundle.

# Router Scroll Position Restoration

You may configure the router to remember and restore scroll position as the user navigates around an application. New navigation events will reset the scroll position, and pressing the back button will restore the previous position.

To turn on restoration in the router configuration:

```
provideRouter(appRoutes,  
  withInMemoryScrolling({scrollPositionRestoration: 'enabled'})  
)
```

# Lazy-Loading Components

Lazy-loading in Angular is a performance optimization technique that delays shipping component source code until it is needed, rather than loading everything upfront during the initial application bootstrap.

This approach reduces the initial bundle size, speeds up the application's startup time, and conserves bandwidth, especially for large applications with multiple features.

# Lazy-Loading via Deferrable Views

The deferrable views **lazily load comments** and all their transitive dependencies via a compile-time transformation. Angular finds all components, directives, and pipes used inside of a `@defer` block, generates dynamic imports, and manages the process of loading and switching between states.

```
@defer (on viewport) {  
  <comment-list/>  
} @loading {  
  Loading...  
} @error {  
  Loading failed :(  
} @placeholder {  
  <!-- A placeholder content to show until the comments load -->  
    
}
```

# Lazy-Loading via Router

```
export const routes: Routes = [  
  {  
    path: '/login',  
    component: LoginComponent  
  },  
  {  
    path: '/signup',  
    loadComponent: () => import('./pages/signup.component').then(c => c.SignupComponent);  
  },  
];
```

# Lazy-Loading Multiple Components

We can accomplish this by importing a **Route array**

```
export const routes: Routes = [  
  {  
    path: 'employees',  
    loadChildren: () => import('./employee.routes').then(r => r.employee_routes);  
  },  
];  
  
export const employee_routes: Routes = [  
  { path: 'list', component: EmployeeListComponent },  
  { path: 'details/:id', component: EmployeeDetailsComponent },  
  { path: 'create', component: CreateEmployeeComponent },  
  { path: 'edit', component: EditEmployeeComponent },  
];
```

# Providing Services Only to Certain Routes

It is possible to add a **providers** array to a route definition.

```
{  
  path: 'employees',  
  providers: [EmployeeService],  
  loadChildren: () => import('./employee.routes').then(r => r.employee_routes);  
},
```


This will make the **EmployeeService** only provided in the routes that reside inside **employee.routes.ts**, meaning only components that are routed in that configuration file will have access to this particular instance of **EmployeeService**.

# Nested Routes

Nested routes is the concept of containing routes within other routes. With nested routes we're able to encapsulate the functionality of parent routes and have that functionality apply to the child routes.

We can have multiple, nested router-outlet. So each area of our application can have their own child components, that also have their own router-outlet.

```
const MY_ROUTES: Routes = [  
  { path: 'parent', component: ParentComponent,  
    children: [  
      { path: 'child', component: ChildComponent }  
    ]  
  }  
];
```



It has an outlet, so its children are rendered in it



# Router Hooks

There are times that we may want to do some actions when changing routes (for example authentication).

Let's say we have a login route and a protected route. We want to only allow the app to go to the protected route if the correct username and password were provided on the login page. In order to do that, we need to **hook into the lifecycle of the router** and ask to be notified when the protected route is being activated. We then can call an authentication service and ask whether or not the user provided the right credentials.

# Guards

Guards allow you to control access to and from a Route/Component.

**canActivate** called when you are serving into the route

**canDeactivate** called when leaving the route.

```
const ROUTES: Routes = [  
  { path: 'my-path',  
    component: MyComponent,  
    canActivate: [Guard1, Guard2..],  
    canMatch: [Guard3, Guard4..],  
    canDeactivate: [Guard5, Guard6..] }  
];
```

# Guard Example

For a reusable guard, generate one with: `ng generate guard MyGuard`

or pass one directly

```
{  
  path: 'admin',  
  canActivate: [(route, state) => inject(LoginService).isLoggedIn()]  
}
```

`canActivate` is used to prevent unauthorized users from accessing certain routes, it returns a `boolean` indicating whether to proceed or not.

# canMatch Guard

The **CanMatch** guard is used to load different components with the same path. If one of the defined guards returns **false**, the route is skipped, and other routes are processed instead.

Example: It can be used to load different components based on the user's role.

```
const routes: Routes = [
  {
    path: 'profile',
    canMatch: [() => inject(AuthService).isRole('admin')],
    loadComponent: () => import('./admin.component').then(c => c.AdminComponent)
  },
  {
    path: 'profile',
    loadComponent: () => import('./user.component') .then(c => c.UserComponent)
  }
];
```

# Dynamic Page Title

Use the Title service to set your page title dynamically as follows:

```
@Component({
  selector: 'app-root',
  template: `...`
})
export class App {
  #title = inject(Title);

  constructor() {
    this.#title.setTitle('Theo');
  }
}
```

# Environment Variables

In the `/src/environments` folder you have one environment file for **development** and another for **production**.

Angular takes care of swapping the environment file for the correct one.

`ng generate environments`

```
"development": {  
  "fileReplacements": [  
    {  
      "replace": "src/environments/environment.ts",  
      "with": "src/environments/environment.development.ts"  
    }  
  ]  
}
```

angular.json



# Web Workers

Web workers lets you run CPU-intensive computations in a background thread, freeing the main thread to update the user interface.

`ng generate web-worker <location>`

```
addEventListener('message', ({ data }) => {  
  const response = { student_id: data.id, grade: 95 }  
  postMessage(response);  
});
```

```
const worker = new Worker(new URL('./app.worker', import.meta.url));  
worker.onmessage = ({ data }) => {  
  results: { student_id: number, grade: number } = data;  
};  
worker.postMessage({ id: 98123 });
```



# File Upload - Angular

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ReactiveFormsModule],
  template: `
    <form [formGroup]="form" (ngSubmit)="submit()">
      <input type="email" [formControl]="form.controls.email"/>
      <input [formControl]="form.controls.avatar" type="file" (change)="onFileSelect($event)" />
      <button [disabled]="form.invalid">Submit</button>
    </form> `
})
export class App {

  form = inject(FormBuilder).nonNullable.group({
    email: ['asaad@miu.edu', Validators.required],
    avatar: ['', Validators.required],
  })
  ... Next page
}
```





# File Upload - Angular (Continued)

```
#file!: File;
#http = inject(HttpClient);

onFileSelect(event: Event) {
  const input = event.target as HTMLInputElement;
  if (input.files!.length > 0) this.#file = input.files![0];
}

submit() {
  const formData = new FormData();
  formData.append('email', this.form.controls.email?.value as string);
  formData.append('avatar', this.#file);

  this.#http.post<{ success: boolean }>('http://localhost:3000/', formData)
    .subscribe(response => {
      this.form.reset();
    })
}
}
```



# File Upload - Express - server.ts

```
import express from 'express';
import cors from 'cors';
const multer from 'multer';

const app = express()

const upload = multer({ dest: 'uploads/' })

app.use(cors())

app.post('/upload', upload.single('avatar'), async (req, res, next) => {
  console.log(req.body)
  console.log(req.file)
  res.json({ success: true })
})

app.listen(3000, () => console.log(`Server is listening on 3000...`))
```



# Deploy for Production

```
> ng build // When you run the ng build command, it creates a /dist folder.
```

- Removes unwanted white space by minifying files.
- Uglifies files by renaming functions and variable names.
- AoT (Ahead-of-Time) compilation

To serve any static resource from the /public folder, set the base for your assets:

```
> ng build --base-href /public/
```