PoiemaWeb

13.21 Angular Forms - Reactive Forms

리액티브 폼과 유효성 검증

#1. 리액티브 폼(Reactive Forms / 모델 기반 폼)이란?

리액티브 폼(모델 기반 폼)은 템플릿이 아닌 컴포넌트 클래스에서 폼 요소의 상태를 관리하는 객체인 폼 모델을 구성하는 방식이다. 리액티브 폼은 템플릿 기반 폼보다 비교적 복잡한 경우 사용한다.

템플릿 기반 폼은 NgForm, NgModel, NgModelGroup 디렉티브를 템플릿 내의 폼 요소 또는 폼 컨트롤 요소에 선언하고 이들 디렉티브가 자신이 적용된 폼 요소 또는 폼 컨트롤 요소에 해당하는 FormGroup, FormControl 인스턴스(폼 모델)를 생성한다. 그리고 NgForm, NgModel, NgModelGroup 디렉티브는 이들 인스턴스(폼 모델)를 폼 요소 또는 폼 컨트롤 요소에 바인딩하여 값이나 유효성 검증 상태를 추적할 수 있었다.

템플릿 기반 폼은 폼 모델을 직접 정의/생성할 수 없고 폼 모델에 직접 접근할 수도 없다. 폼 모델에 접근하기 위해서는 NgForm, NgModel, NgModelGroup 디렉티브가 생성한 폼 모델을 템플릿 참조 변수에 할당하여야 한다.

NgForm, NgModel, NgModelGroup 디렉티브가 생성한 폼 모델을 템플릿 참조 변수에 할당하여 값이나 유효성 검증 상태에 접근할 수 있다.

리액티브 폼(모델 기반 폼)은 컴포넌트 클래스에서 폼 요소의 값 및 유효성 검증 상태를 관리하는 자바스크립트 객체인 폼 모델(FormGroup, FormControl, FormArray)을 직접 정의/생성한다. 그리고 form* 접두사가 붙은 디렉티브(formGroup, formGroupName, formControlName, formArrayName)를 사용하여 템플릿의 폼 요소와 컴포넌트 클래스의 폼 모델을 프로퍼티 바인딩으로 연결한다. 다시 말해 컴포넌트 클래스 내부에서 정의/생성한 폼 모델에 직접 접근하여 데이터 모델을 폼 모델에 반영하고 템플릿의 폼 컨트롤 요소의 상태 변화를 관찰(observe)하고 변화에 대응한다.

리액티브 폼은 FormGroup, FormControl, FormArray 클래스를 중심으로 동작한다. 이들을 사용하기 위해서 @angular/forms 패키지의 ReactiveFormsModule을 애플리케이션 모듈에 추가한다.

TYPESCRIPT

```
// app.module.ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, ReactiveFormsModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

#2. 리액티브 폼의 중심 클래스와 디렉티브

2.1 FormGroup 클래스와 formGroup/formGroupName 디렉티브

FormGroup 클래스가 생성하는 FormGroup 인스턴스는 자신의 자식인 FormControl 인스턴스 또는 FormArray 인스턴스들을 그룹화하여 관리하기 위한 최상위 컨테이너이다. FormControl 또는 FormArray 인스턴스와 같은 자식 폼 모델 인스턴스들을 하나의 객체로 그룹화하여 모든 자식 폼 모델 인스턴스의 값과 유효성 상태를 관리한다. 만약 유효성을 검증할 때 자식 폼 모델 인스턴스 중 하나라도 유효하지 않다면 FormGroup은 유효하지 않게 된다.

TYPESCRIPT

```
const myFormGroup = new FormGroup({
  // 자식 폼 모델 인스턴스
});
```

FormGroup 인스턴스는 템플릿의 폼 요소와 대응한다. 템플릿 기반 폼에서는 NgForm 디렉티브를 사용하여 자신이 적용된 폼 요소에 해당하는 FormGroup 인스턴스를 생성하였다. 리액티브 폼에서는 컴포넌트 클래스에 FormGroup 인스턴스를 직접 생성하고 formGroup 디렉티브를 사용하여 FormGroup 인스턴스와 템플릿의 폼 요소를 바인딩한다.

TYPESCRIPT

```
// app.component.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup } from '@angular/forms';

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="userForm" novalidate></form>
  `
})
export class AppComponent implements OnInit {

  userForm: FormGroup;

  ngOnInit() {
```

```
    this.userForm = new FormGroup({});
    console.log(this.userForm); // FormGroup
  }
}
```

reactive-form-1
By ungmo2

폼 요소에 선언한 novalidate 어트리뷰트는 표준 HTML 유효성 검증을 방지한다. 리액티브 폼이 제공하는 유효성 검증기를 사용할 것이므로 novalidate 어트리뷰트를 선언하도록 한다.

FormGroup 인스턴스는 폼 요소 내부의 폼 컨트롤 요소들을 그룹화하기 위해 또다른 FormGroup 인스턴스를 가질 수 있다. formGroupName 디렉티브는 FormGroup 인스턴스의 자식 FormGroup 인스턴스와 폼 컨트롤 요소 그룹을 바인딩한다.

TYPESCRIPT

```
// app.component.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup } from '@angular/forms';

@Component({
```

```
  selector: 'app-root',
  template: `
    <form [formGroup]="userForm" novalidate>
      <div FormGroupName="formControls"></div>
    </form>
  `
})
export class AppComponent implements OnInit {

  userForm: FormGroup;

  ngOnInit() {
    this.userForm = new FormGroup({
      formControls: new FormGroup({})
    });
    console.log(this.userForm);
  }
}
```

reactive-form-2
By ungmo2

Run Project

formGroupName 디렉티브에는 문자열을 할당하여야 한다. 이 문자열은 FormGroup 인스턴스를 값으로 하는 프로퍼티 이름이다.

```
// app.component.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup } from '@angular/forms';

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="userForm" novalidate>
      <div FormGroupName="formControls"></div>
    </form>
  `
})
export class AppComponent implements OnInit {

  userForm: FormGroup;

  ngOnInit() {
    this.userForm = new FormGroup({
      formControls: new FormGroup({})
    });
    console.log(this.userForm);
  }
}
```

formGroupName 디렉티브

프로퍼티 바인딩 문법으로 문자열을 할당하려면 따옴표를 연이어 사용하여야 하므로 프로퍼티 바인딩의 대괄호를 생략하고 문자열을 할당하였다. 이는 이후 등장하는 모든 form*Name 디렉티브에 적용된다.

HTML

```
<div [FormGroupName]="'formControls'"></div>
<!-- 위아래 두 표현은 동치이다 -->
<div FormGroupName="formControls"></div>
```

2.2 FormControl 클래스와 formControlName 디렉티브

FormControl 클래스가 생성하는 FormControl 인스턴스는 폼을 구성하는 기본 단위로서 폼 컨트롤 요소의 값이나 유효성 검증 상태를 추적하고 뷰와 폼 모델을 동기화된 상태로 유지한다.

TYPESCRIPT

```
const myFormControl = new FormControl('initial value');
```

FormControl 인스턴스는 템플릿의 개별 폼 컨트롤 요소와 대응한다. 템플릿 기반 폼에서는 NgModel 디렉티브가 자신이 적용된 폼 컨트롤 요소에 해당하는 FormControl 인스턴스를 생성하였다. 리액티브 폼에서는 컴포넌트 클래스에서 FormControl 인스턴스를 직접 생성하고 formControlName 디렉티브를 사용하여 FormControl 인스턴스와 템플릿의 폼 컨트롤 요소를 바인딩한다.

TYPESCRIPT

```
// app.component.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="userForm" novalidate>
      <div>
        <input type="text" formControlName="userid" placeholder="user id">
      </div>
      <div formGroupName="passwordGroup">
        <div>
          <input type="password" formControlName="password" placeholder="pas
sword">
        </div>
        <div>
          <input type="password" formControlName="confirmPassword" placehold
er="confirm password">
        </div>
      </div>
    </form>
```

```
    <pre>{{ userForm.value | json }}</pre>
    <pre>{{ userForm.status }}</pre>
  `
})
export class AppComponent implements OnInit {

  userForm: FormGroup;

  ngOnInit() {
    this.userForm = new FormGroup({
      userid: new FormControl(''),
      passwordGroup: new FormGroup({
        password: new FormControl(''),
        confirmPassword: new FormControl('')
      })
    });
    console.log(this.userForm);
  }
}
```

FormControl은 폼 요소의 자식인 폼 컨트롤 요소를 위해 사용하기도 하지만, 폼 요소 없이 단독으로 사용할 수도 있다. [Reactive Programming과 RxJS] 옵저버블 이벤트 스트림에서 살펴본 바와 같이 input 요소의 이벤트는 FormControl의 valueChanges 프로퍼티에 의해 옵저버블 스트림으로 변환된다.

TYPESCRIPT

```
// app.component.ts
...
@Component({
  selector: 'app-root',
  template: `
    <h2>Observable Events</h2>
    <input type="text" placeholder="Enter user id" [formControl]="serchInput">
    <pre>{{ githubUser | json }}</pre>
  `
```

```
})
export class ObservableEventHttpComponent implements OnInit , OnDestroy {
  // Angular 리액티브 폼
  serchInput: FormControl = new FormControl('');
  githubUser: GithubUser;
  subscription: Subscription;

// 서버와의 통신을 위해 HttpClient를 의존성 주입한다.
constructor(private http: HttpClient) { }

ngOnInit() {
    // 폼 컨트롤 값의 상태를 옵저버블 스트림으로 수신한다.
    this.subscription = this.searchInput.valueChanges
      .pipe(
        // 옵저버블이 방출하는 데이터를 수신하는 시간을 지연시킨다.
        debounceTime(500),
        // 새로운 옵저버블을 생성한다.
        switchMap((userId: string) => this.getGithubUser(userId))
      )
      // 옵저버블을 구독한다.
      .subscribe(
        user => this.githubUser = user,
        error => console.log(error)
      );
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
  ...
}
```

2.3 FormArray 클래스와 formArrayName 디렉티브

FormArray 클래스가 생성하는 FormArray 인스턴스는 자바스크립트의 배열과 유사하게 FormControl 인스턴스들을 그룹화하여 관리한다. FormArray는 폼 컨트롤 요소가 동적으로 생성되어 그 개수가 변화할 때 사용한다.

TYPESCRIPT

```
const myFormArray = new FormArray([
  new FormControl(''),
  new FormControl('')
]);
```

formArrayName 디렉티브는 FormArray 인스턴스를 DOM 요소에 바인딩한다.

TYPESCRIPT

```
// app.component.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, FormArray } from '@angular/forms';

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="userForm" novalidate>
      <div formArrayName="hobbies">
        <!-- (1) -->
        <div *ngFor="let hobby of hobbies.controls; let i=index">
          <input type="text" [formControlName]="i">
        </div>
      </div>
    </form>
    <pre>{{ userForm.value | json }}</pre>
    <pre>{{ userForm.status }}</pre>
  `
})
export class AppComponent implements OnInit {

  userForm: FormGroup;

  ngOnInit() {
    this.userForm = new FormGroup({
      hobbies: new FormArray([
        new FormControl(''),
        new FormControl('')
      ])
    });
    console.log(this.userForm);
  }

  // 2) 템플릿에서 폼 모델에 접근할 수 있도록 컴포넌트 클래스에 getter를 정의한다.
  get hobbies(): FormArray {
    return this.userForm.get('hobbies') as FormArray;
  }
}
```

1) ngFor 디렉티브를 사용하여 FormArray의 요소의 개수만큼 순회하며 폼 컨트롤 요소를 생성한다. 이때 폼 컨트롤 요소의 formControlName 디렉티브에 인덱스 i를 할당한다. 주의할 것은 인덱스 i는 변수이므로 프로퍼티 바인딩을 사용하여야 한다.

2) 템플릿이 FormArray에 접근할 수 있도록 getter를 정의하였다. 이 getter를 통해 템플릿은 컴포넌트 클래스의 hobbies 프로퍼티에 접근할 수 있게 되었다. 이 getter는 FormArray 타입의 객체를 반환하는데 FormArray 타입의 controls 프로퍼티를 사용하여 개별 요소에 접근할 수 있다.

TYPESCRIPT

```
...
@Component({
  selector: 'app-root',
  template: `
        ...
```

```
        <!-- 컴포넌트 클래스의 getter hobbies를 통해 폼 모델에 접근 -->
        <div *ngFor="let hobby of hobbies.controls; let i=index">
        ...
  `
})
```

#3. 리액티브 폼 유효성 검증

템플릿 기반 폼은 유효성 검증이 필요한 템플릿의 폼 컨트롤 요소에 required, pattern과 같은 빌트인 검증기(Built-in validator)를 선언한다.

리액티브 폼은 템플릿의 폼 컨트롤 요소에 빌트인 검증기를 선언하지 않고 컴포넌트 클래스 내부에서 생성한 FormControl에 추가한다. FormControl에 추가된 검증기는 템플릿의 폼 컨트롤 요소의 상태가 변화할 때마다 호출된다.

리액티브 폼에서 사용 가능한 빌트인 검증기는 Validators 클래스에 정적 메소드로 정의되어 있다.

TYPESCRIPT

```
class Validators {
  static min(min: number): ValidatorFn
  static max(max: number): ValidatorFn
  static required(control: AbstractControl): ValidationErrors|null
  static requiredTrue(control: AbstractControl): ValidationErrors|null
  static email(control: AbstractControl): ValidationErrors|null
  static minLength(minLength: number): ValidatorFn
  static maxLength(maxLength: number): ValidatorFn
  static pattern(pattern: string|RegExp): ValidatorFn
  static nullValidator(c: AbstractControl): ValidationErrors|null
  static compose(validators: (ValidatorFn|null|undefined)[]|null): Validator
Fn|null
  static composeAsync(validators: (AsyncValidatorFn|null)[]): AsyncValidator
Fn|null
}
```

아래의 예제를 살펴보자.

TYPESCRIPT

```
// app.component.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, FormArray, Validators } from '@angular/form
s';

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="userForm" novalidate>
      <div>
        <input type="text" formControlName="userid" placeholder="user id">
      </div>
      <div formGroupName="passwordGroup">
        <div>
          <input type="password" formControlName="password" placeholder="pas
sword">
        </div>
        <div>
          <input type="password" formControlName="confirmPassword" placehold
er="confirm password">
        </div>
      </div>
    </form>
    <pre>{{ userForm.value | json }}</pre>
    <pre>{{ userForm.status }}</pre>
  `
})
export class AppComponent implements OnInit {

  userForm: FormGroup;

  ngOnInit() {
    this.userForm = new FormGroup({
      /* FormControl 생성자 함수의 두번째 인자에 검증기를 전달한다.
         2개 이상의 검증기를 사용하는 경우, 배열로 검증기를 추가한다. 검증기는 템플릿의 폼
 컨트롤 요소의 상태가 변화할 때 마다 호출된다. */
      userid: new FormControl('', [
        Validators.required,
        Validators.pattern('[a-zA-Z0-9]{4,10}')
      ]),
```

```
      passwordGroup: new FormGroup({
        // FormControl 생성자 함수의 두번째 인자에 검증기를 전달한다.
        password: new FormControl('', Validators.required),
        confirmPassword: new FormControl('', Validators.required)
      }),
      hobbies: new FormArray([
        new FormControl(''),
        new FormControl('')
      ])
    });
  }
}
```

reactive-form-5
By ungmo2

리액티브 폼은 템플릿의 컴포넌트 클래스 내부에서 생성한 FormControl에 검증기를 추가한다. 검증기는 FormControl 생성자의 두번째 인자에 전달한다. 2개 이상의 검증기를 사용하는 경우, 배열로 검증기

를 추가한다. 검증기는 템플릿의 폼 컨트롤 요소의 상태가 변화할 때 마다 호출된다.

4. 사용자 정의 검증기(Custom validator)

빌트인 검증기는 사용이 간편하지만 기본적인 검증 기능만을 제공하므로 복잡한 애플리케이션의 요구 사항을 충족시키기 어려운 경우가 있다. Angular는 사용자 정의 검증기를 정의할 수 있으며 템플릿 기반 폼과 리액티브 폼 모두에 사용할 수 있다. 사용자 정의 검증기의 정의 방법에 대해 살펴보도록 하자.

사용자 정의 검증기는 재사용을 위해 외부 클래스로 분리하는 것이 일반적이다. 패스워드와 확인 패스워드가 일치하는지 검증하는 사용자 정의 검증기를 작성해보자.

TYPESCRIPT

```
// password-validator.ts
import { AbstractControl } from '@angular/forms';

export class PasswordValidator {
  static match(form: AbstractControl) {
    // 매개변수로 전달받은 검증 대상 폼 모델에서 password와 confirmPassword을 취득
    const password = form.get('password').value;
    const confirmPassword = form.get('confirmPassword').value;

    // password와 confirmPassword의 값을 비교한다.
    if (password !== confirmPassword) {
      // 검증에 실패한 경우, 에러 객체를 반환한다.
      return { match: { password, confirmPassword }};
    } else {
      // 검증에 성공한 경우, null을 반환한다.
      return null;
    }
  }
}
```

사용자 정의 검증기는 클래스의 정적 메소드로 정의한다. 이때 메소드의 매개변수는 검증 대상 폼 모델이다. 다시 말해 폼 모델에 사용자 정의 검증기를 선언하면 해당 사용자 정의 검증기의 매개변수에 폼 모델이 전달된다.

PasswordValidator 클래스의 정적 메소드 match는 패스워드와 확인 패스워드를 입력하는 폼 컨트롤 요소를 그룹화한 FormGroup의 인스턴스 passwordGroup에 적용할 것이기 때문에 매개변수 타입을 AbstractControl로 지정하였다. 만약 사용자 정의 검증기가 폼 컨트롤 요소에 적용된다면 매개변수 타입을 FormControl로 지정하여도 된다.

매개변수로 전달받은 검증 대상 폼 모델(위 예제의 경우, passwordGroup)에서 get 메소드를 사용하여 password와 confirmPassword을 취득하고 두 값을 비교한다. 두 값이 불일치하는 경우, 에러 내용을 나타내는 에러 객체를 반환한다. 이 에러 객체는 템플릿에서 userForm.controls.passwordGroup.errors?.match로 참조할 수 있다.

두 값이 일치하여 오류가 발생하지 않는 경우, null을 반환한다. 이때 passwordGroup.errors는 null이 된다.

사용자 정의 검증기는 빌트인 검증기와 동일한 방식으로 사용한다.

TYPESCRIPT

```
// app.component.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, FormArray, Validators } from '@angular/form
s';
import { PasswordValidator } from './password-validator';

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="userForm" novalidate>
      <div>
        <input type="text" formControlName="userid" placeholder="user id">
      </div>
      <div formGroupName="passwordGroup">
        <div>
          <input type="password" formControlName="password" placeholder="pas
sword">
        </div>
        <div>
          <input type="password" formControlName="confirmPassword" placehold
er="confirm password">
        </div>
      </div>
    </form>
    <pre>{{ userForm.value | json }}</pre>
```

```
      <pre>{{ userForm.status }}</pre>
      <pre>{{ userForm.controls.passwordGroup.errors?.match | json }}</pre>
    `
  })
  export class AppComponent implements OnInit {

    userForm: FormGroup;

    ngOnInit() {
      this.userForm = new FormGroup({
        userid: new FormControl('', [
          Validators.required,
          Validators.pattern('[a-zA-Z0-9]{4,10}')
        ]),
        passwordGroup: new FormGroup({
          password: new FormControl('', Validators.required),
          confirmPassword: new FormControl('', Validators.required)
        }, PasswordValidator.match) // 사용자 정의 검증기 적용
      });
    }
  }
```

5. 리액티브 폼 유효성 검증 실습

리액티브 폼을 사용하여 회원 가입 폼을 작성해보자. 이 예제는 부트스트랩을 사용할 것이므로 부트스트랩을 설치하도록 한다.

BASH

```
$ ng new reactive-form-exam -st
$ cd reactive-form-exam
$ npm install bootstrap@3.3.7
```

설치가 완료되었으면 부트스트랩을 임포트하여야 한다. 부트스트랩은 모든 컴포넌트에 적용되어야 하므로 angular.json를 아래와 같이 수정한다.

JSON

```
{
  ...
  "apps": [
  ...
    "styles": [
      "../node_modules/bootstrap/dist/css/bootstrap.min.css",
      "styles.css"
    ],
  ...
}
```

다음은 회원 가입 폼 컴포넌트를 생성한다.

BASH

```
$ ng generate component user-form
```

루트 컴포넌트를 아래와 같이 수정한다.

TYPESCRIPT

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: '<user-form></user-form>'
})
export class AppComponent {}
```

리액티브 폼을 사용하기 위해 루트 모듈에 ReactiveFormsModule을 추가한다.

TYPESCRIPT

```typescript
// app.module.ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { UserFormComponent } from './user-form/user-form.component';

@NgModule({
  declarations: [
    AppComponent,
    UserFormComponent
  ],
  imports: [
    BrowserModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

이제 아래와 같이 회원 가입 폼 템플릿을 작성한다.

HTML

```html
<!-- user-form/user-form.component.html -->
<div class="container">
  <h2>Reactive forms Exam</h2>
  <form [formGroup]="userForm" (ngSubmit)="onSubmit()" novalidate>
    <div class="form-group">
      <label for="userid">User id</label>
      <input type="text" name="userid" class="form-control"
        formControlName="userid">
      <em *ngIf="userid.errors?.required && userid.touched" class="alert">
        userid를 입력하세요!
      </em>
      <em *ngIf="userid.errors?.pattern && userid.touched" class="alert">
        userid는 영문 또는 숫자로 4자리 이상 10이하로 입력하세요!
```

```
        </em>
      </div>
      <div formGroupName="passwordGroup">
        <div class="form-group">
          <label for="password">Password</label>
          <input type="password" name="password" class="form-control"
            formControlName="password">
          <em *ngIf="password.errors?.required && password.touched" class="ale
rt">
            password를 입력하세요!
          </em>
        </div>
        <div class="form-group">
          <label for="confirmPassword">Confirm password</label>
          <input type="password" name="confirmPassword" class="form-control"
            formControlName="confirmPassword">
          <em *ngIf="confirmPassword.errors?.required && confirmPassword.touch
ed" class="alert">
            password를 입력하세요!
          </em>
          <em *ngIf="passwordGroup.errors?.match
              && confirmPassword.touched
              && !confirmPassword.errors?.required"
              class="alert">
            password가 일치하지 않습니다!
          </em>
        </div>
      </div>
      <div formArrayName="hobbies">
        <div *ngFor="let hobby of hobbies.controls; let i=index">
          <div class="form-group">
            <label for="hobby{{i}}">hobby {{i}}</label>
            <input type="text" name="hobby{{i}}" class="form-control"
              [formControlName]="i">
          </div>
        </div>
      </div>
      <button type="submit" class="btn btn-success"
        [disabled]="userForm.invalid">Submit</button>
    </form>
    <pre>{{ userForm.value | json }}</pre>
```

```
  <pre>{{ userForm.status }}</pre>
</div>
```

컴포넌트 CSS는 아래와 같다.

CSS

```
/* user-form/user-form.component.css */
.alert { color: red; }
```

폼 컨트롤 요소에 form* 디렉티브를 선언하여 템플릿의 폼 요소와 폼 모델을 프로퍼티 바인딩으로 연결한다. 모든 폼 컨트롤 요소의 유효성 검증은 컴포넌트 클래스의 폼 모델이 담당할 것이며 모든 요소의 유효성 검증에 성공한 상태(userForm.valid가 true 또는 userForm.invalid가 false)라면 submit 버튼이 활성화된다.

컴포넌트 클래스는 아래와 같다.

TYPESCRIPT

```
// user-form/user-form.component.ts
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, FormArray, Validators } from '@angular/form
s';
import { PasswordValidator } from '../password-validator';

@Component({
  selector: 'user-form',
  templateUrl: './user-form.component.html',
  styleUrls: ['./user-form.component.css']
})
export class UserFormComponent implements OnInit {
  userForm: FormGroup;

  ngOnInit() {
    this.userForm = new FormGroup({
      userid: new FormControl('', [
        Validators.required,
        Validators.pattern('[a-zA-Z0-9]{4,10}')
      ]),
```

```
      passwordGroup: new FormGroup({
        password: new FormControl('', Validators.required),
        confirmPassword: new FormControl('', Validators.required)
      }, PasswordValidator.match),
      hobbies: new FormArray([
        new FormControl(''),
        new FormControl('')
      ])
    });
  }

  // 템플릿에서 폼 모델에 접근할 수 있도록 컴포넌트 클래스에 getter를 정의한다.
  get userid() {
    return this.userForm.get('userid');
  }

  get passwordGroup() {
    return this.userForm.get('passwordGroup');
  }

  get password() {
    return this.userForm.get('passwordGroup.password');
  }

  get confirmPassword() {
    return this.userForm.get('passwordGroup.confirmPassword');
  }

  get hobbies(): FormArray {
    return this.userForm.get('hobbies') as FormArray;
  }

  onSubmit() {
    console.log(this.userForm);
    this.userForm.reset();
  }
}
```

컴포넌트 클래스에서 폼 요소의 값 및 유효성 검증 상태를 관리하는 자바스크립트 객체인 폼 모델 userForm을 정의, 생성하였다. 이 폼 모델은 FormGroup의 인스턴스로서 폼의 **데이터 구조를 정의**하

며 모든 자식 인스턴스들의 값이나 유효성 검증 상태를 추적하고 뷰와 폼 모델을 동기화된 상태로 유지한다.

그리고 form* 접두사가 붙은 디렉티브(formGroup, formGroupName, formControlName, formArrayName)를 사용하여 템플릿의 폼 요소와 폼 모델을 프로퍼티 바인딩으로 연결한다.

템플릿 기반 폼은 폼 모델에 직접 접근할 수 없지만 리액티브 폼은 폼 모델을 직접 정의, 생성하고 조작한다. 즉, 컴포넌트 클래스 내부에서 데이터 모델과 폼 모델에 직접 접근하여 데이터 모델을 폼 모델에 반영하고 템플릿 폼 컨트롤 요소의 상태 변화를 관찰(observe)하고 변화에 대응한다.

리액티브 폼은 템플릿의 컴포넌트 클래스 내부에서 생성한 FormControl에 검증기를 추가한다. 검증기는 FormControl 생성자의 두번째 인자에 전달한다. 2개 이상의 검증기를 사용하는 경우, 배열로 검증기를 추가한다. 검증기는 템플릿의 폼 컨트롤 요소의 상태가 변화할 때 마다 호출된다.

사용자 정의 검증기 PasswordValidator는 아래와 같다.

TYPESCRIPT

```
// password-validator.ts
import { AbstractControl } from '@angular/forms';

export class PasswordValidator {

  static match(form: AbstractControl) {
    const password = form.get('password').value;
    const confirmPassword = form.get('confirmPassword').value;

    if (password !== confirmPassword) {
      return { match: { password, confirmPassword }};
    } else {
      return null;
    }
  }
}
```

템플릿 기반 폼은 폼 모델을 직접 정의/생성할 수 없고 폼 모델에 직접 접근할 수도 없다. 폼 모델에 접근하기 위해서는 NgForm, NgModel, NgModelGroup 디렉티브가 생성한 폼 모델을 템플릿 참조 변수에 할당하여야 한다.

HTML

```
<input type="text" name="userid" ngModel #userid="ngModel">
<pre>{{ userid.value }}</pre>
```

하지만 리액티브 폼의 경우, 템플릿은 컴포넌트 클래스의 프로퍼티에 접근할 수 있으므로 컴포넌트 클래스에서 정의/생성한 폼 모델에 접근할 수 있다. **템플릿에서 폼 모델에 간편하게 접근할 수 있도록 컴포넌트 클래스에 getter를 정의한다.**

AbstractControl 클래스의 get 메소드는 폼 모델의 자식 인스턴스를 검색할 때 사용한다.

TYPESCRIPT

```
get userid() {
  return this.userForm.get('userid');
}

get passwordGroup() {
  return this.userForm.get('passwordGroup');
}

get password() {
  return this.userForm.get('passwordGroup.password');
}

get confirmPassword() {
  return this.userForm.get('passwordGroup.confirmPassword');
}

get hobbies(): FormArray {
  return this.userForm.get('hobbies') as FormArray;
}
```

템플릿은 getter를 참조하여 폼 모델이 관리하는 폼 컨트롤 요소의 값이나 유효성 검증 상태를 추적할 수 있다.

HTML

```
<em *ngIf="passwordGroup.errors?.match && confirmPassword.touched && !confir
mPassword.errors?.required" class="alert">
```

```
    password가 일치하지 않습니다!
  </em>
```

/src/app/user-form.component.ts

<> EDITOR

PREVIEW

6. FormBuilder

지금까지 살펴본 방식으로도 리액티브 폼을 구성하고 유효성 검증을 실행할 수 있다. 하지만 Angular는 더욱 간편한 방법으로 리액티브 폼을 구성할 수 있는 FormBuilder를 제공한다. FormBuilder를 사용하면 FormGroup, FormControl, FormArray와 같은 클래스를 사용하여 인스턴스를 생성하는 번거로움을 줄일 수 있어서 코드가 간결해지고 가독성이 좋아진다. 이는 복잡한 폼을 보다 쉽게 구성할 수 있도록 도우며 유지보수에도 유리하다.

위에서 작성한 리액티브 폼 유효성 검증 실습 예제를 FormBuilder를 사용하여 리팩토링해보자.

TYPESCRIPT

```
// user-form/user-form.component.ts
...
  // FormBuilder를 주입받는다.
  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.userForm = this.fb.group({
      userid: ['', [
        Validators.required,
        Validators.pattern('[a-zA-Z0-9]{4,10}')
      ]],
      passwordGroup: this.fb.group({
        password: ['', Validators.required],
        confirmPassword: ['', Validators.required]
      }, { validator: PasswordValidator.match }),
      hobbies: this.fb.array(['', ''])
    });
  }
...
```

FormBuilder를 사용하기 위해서는 FormBuilder 클래스의 인스턴스를 주입받는다. new FormGroup()으로 생성하였던 폼 모델은 주입받은 FormBuilder의 인스턴스를 사용하여 this.fb.group()와 같이 생성한다.

FormBuilder를 사용하지 않는 경우, 아래와 같이 생성자의 2번째 인자에 검증기를 추가하였다.

TYPESCRIPT

```
    passwordGroup: new FormGroup({
      password: new FormControl('', Validators.required),
```

```
    confirmPassword: new FormControl('', Validators.required)
  }, PasswordValidator.match),
```

주의하여야 할 것은 FormBuilder를 사용하는 경우, 아래와 같이 생성자의 2번째 인자에 옵션 객체를 생성하여 검증기를 추가하여야 한다.

TYPESCRIPT

```
  passwordGroup: this.fb.group({
    password: ['', Validators.required],
    confirmPassword: ['', Validators.required]
  }, { validator: PasswordValidator.match }),
```

Reference

- FormGroup 클래스
- formGroup 디렉티브
- formGroupName 디렉티브

- FormControl 클래스
- formControlName 디렉티브
- FormArray 클래스
- formArrayName 디렉티브
- Validators
- ValidatorFn
- FormBuilder