PoiemaWeb

13.20 Angular Forms - Template-driven Forms
템플릿 기반 폼과 유효성 검증

#1. 템플릿 기반 폼(Template-driven Forms)이란?

템플릿 기반 폼은 컴포넌트 템플릿에서 디렉티브를 사용하여 폼을 구성하는 방식으로 각 필드의 형식, 유효성 검증 규칙을 모두 템플릿에서 정의한다. 비교적 간단한 폼에 사용한다.

템플릿 기반 폼은 NgForm, NgModel, NgModelGroup 디렉티브를 중심으로 동작한다. 이들을 사용하기 위해서 @angular/forms 패키지의 FormsModule을 루트 모듈에 추가한다.

TYPESCRIPT

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

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

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

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

템플릿 기반 폼의 중심인 NgForm, NgModel, NgModelGroup 디렉티브에 대해 살펴보도록 하자.

2. 템플릿 기반 폼의 중심 디렉티브

2.1 NgForm 디렉티브

NgForm 디렉티브는 템플릿 기반 폼 전체를 관리하는 디렉티브이다. 루트 모듈에 FormsModule을 추가하면 NgForm 디렉티브를 선언하지 않아도 모든 form 요소에 NgForm 디렉티브가 자동으로 적용되어 템플릿 기반 폼으로 동작한다. 이제 HTML 표준 폼이 제공하는 유효성 검증 기능을 사용하지 않고 템플릿 기반 폼이 제공하는 유효성 검증 기능을 사용할 것이므로 HTML 표준 폼의 유효성 검증 기능을 비활성하는 `novalidate` 어트리뷰트를 추가한다.

HTML

```
<form novalidate></form>
```

폼 요소에 자동으로 적용되는 NgForm 디렉티브의 적용을 취소하려면 폼 요소에 ngNoForm을 추가한다. ngNoForm이 적용되면 HTML 표준 폼으로 동작한다.

HTML

```
<form ngNoForm></form>
```

HTML 표준 폼은 submit 버튼을 클릭하면 폼 데이터를 서버로 전송하고 페이지를 전환하지만, NgForm 디렉티브가 적용된 템플릿 기반 폼은 submit 이벤트를 인터셉트하여 폼 데이터를 서버로 전송하고 페이지를 전환하는 submit 이벤트의 기본 동작을 막는다. 따라서 템플릿 기반 폼에서는 submit 이벤트 대신 사용자가 폼을 제출(submit)했을 때 NgForm 디렉티브가 방출하는 ngSubmit 이벤트를 사용한다.

HTML

```
<form (ngSubmit)="onNgSubmit()" novalidate></form>
```

템플릿 기반 폼에도 템플릿 참조 변수를 사용할 수 있다. 참조 변수에는 ngForm을 할당하여 참조 변수가 네이티브 DOM이 아닌 NgForm 인스턴스를 가리키도록 한다.

HTML

```
<form #f="ngForm" (ngSubmit)="onNgSubmit(f)" novalidate></form>
```

NgForm 인스턴스는 NgForm 디렉티브가 생성하는 인스턴스로서 폼 전체를 관리한다. NgForm 디렉티브는 NgForm 인스턴스를 생성할 때, 자신이 적용된 폼 요소(별도 설정이 없는 경우 폼 요소에 NgForm 디렉티브가 자동 적용된다)의 **값이나 유효성 검증 상태를 추적할 수 있는 기능**을 제공하는 FormGroup 인스턴스를 생성한 후, NgForm 인스턴스의 프로퍼티로 추가한다. 이렇게 생성된 NgForm 인스턴스를 템플릿 참조 변수에 할당하는 것으로 참조 변수가 네이티브 DOM이 아닌 NgForm 인스턴스를 가리키도록 한다.

이제 템플릿 참조 변수 f는 NgForm 인스턴스를 바인딩하였고 해당 폼 요소의 값이나 유효성 검증 상태를 추적할 수 있게 되었다. 물론 템플릿 참조 변수는 이벤트 핸들러에 인자로 전달할 수도 있다.

그런데 폼 요소는 자기 자신만 존재해서는 의미가 없고 자식 요소로 폼 컨트롤 요소를 가질 때 의미가 있다.

HTML

```
<form #f="ngForm" (ngSubmit)="onNgSubmit(f)" novalidate>
  <input type="text" name="userid" placeholder="userid">
```

```
  ...
</form>
```

NgForm 디렉티브는 폼 요소의 자식 폼 컨트롤 요소 중에서 `NgModel` 디렉티브가 적용된 요소를 탐색하여 FormGroup 인스턴스에 추가한다.

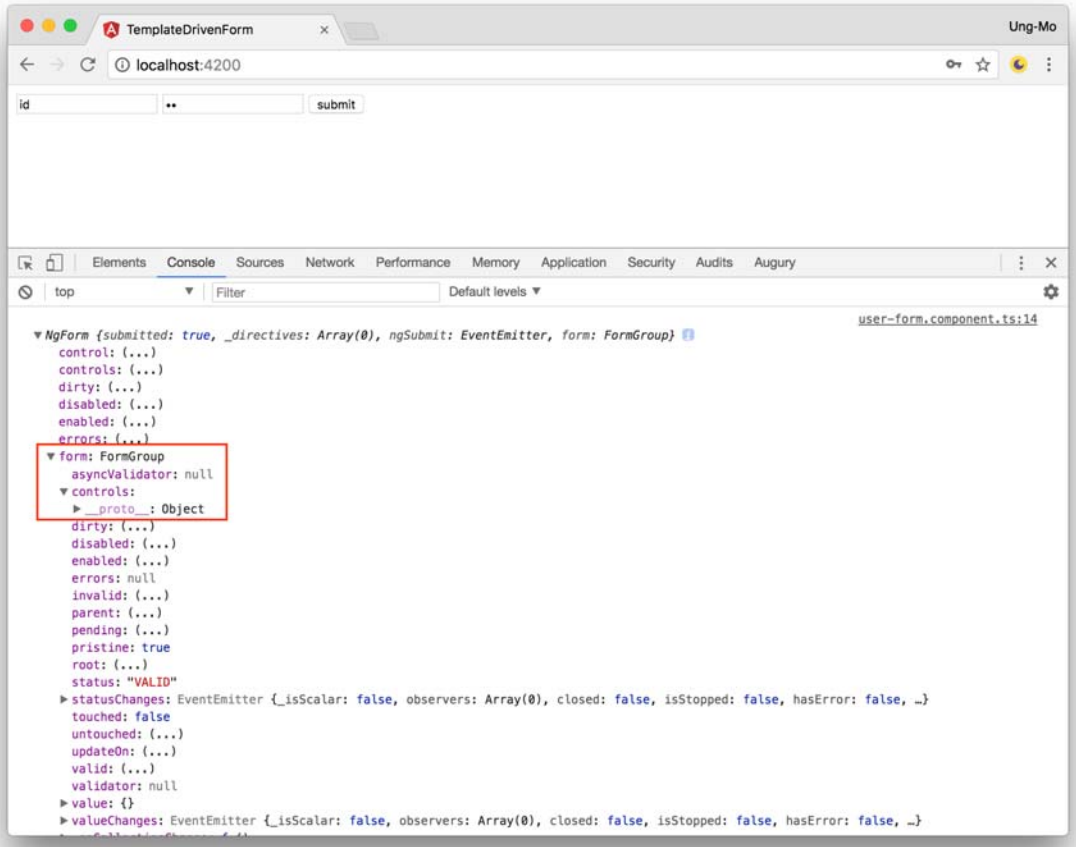
간단한 회원 가입 폼을 작성하여 템플릿 기반 폼에 대해 살펴보도록 하자.

TYPESCRIPT

```
// user-form.component.ts
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'user-form',
  template: `
    <form #userForm="ngForm" (ngSubmit)="onNgSubmit(userForm)" novalidate>
      <input type="text" name="userid" placeholder="userid">
      <input type="password" name="password" placeholder="password">
      <button>submit</button>
    </form>
  `
})
export class UserFormComponent {
  onNgSubmit(userForm: NgForm) {
    console.log(userForm);
  }
}
```

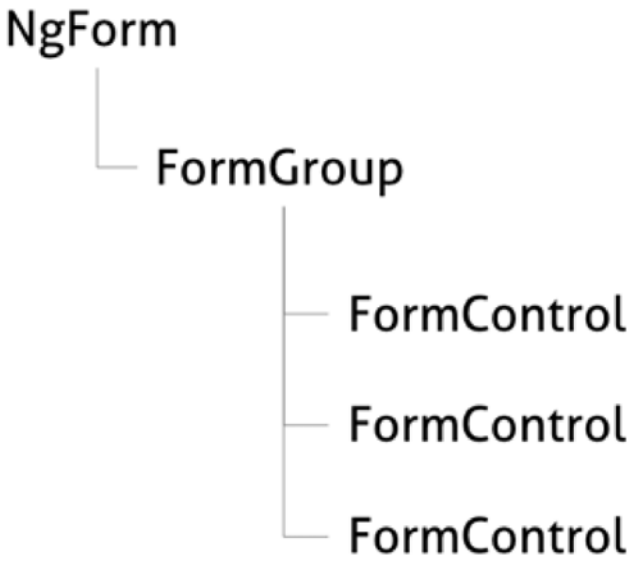
위 예제를 살펴보면 폼 요소의 자식 폼 컨트롤 요소 중에 NgModel 디렉티브가 적용된 요소가 없다. 따라서 NgForm 디렉티브는 어떠한 자식 폼 컨트롤 요소도 FormGroup 인스턴스에 추가하지 않는다. 다시 말해 어떠한 자식 폼 컨트롤 요소도 FormGroup 인스턴스에 추가되지 않았으므로 FormGroup 인스턴스가 제공하는 값이나 유효성 검증 상태 추적 기능을 사용할 수 없다.



NgForm 인스턴스의 프로퍼티

NgModel 디렉티브는 자신이 적용된 폼 컨트롤 요소의 값이나 유효성 검증 상태의 추적 기능을 제공하는 FormControl 인스턴스를 생성한다. 이 FormControl 인스턴스는 FormGroup 인스턴스의 프로퍼티로 추가된다.

앞서 살펴본 FormGroup 인스턴스는 자신의 자식인 FormControl 인스턴스들을 하나의 객체로 그룹화하여 관리하기 위한 최상위 컨테이너로서 모든 FormControl 인스턴스의 값과 유효성 상태를 관리한다. 만약 유효성을 검증할 때 FormControl 인스턴스 중 하나라도 유효하지 않다면 FormGroup은 유효하지 않은 상태인 invalid 상태가 된다.



템플릿 기반 폼의 인스턴스 구조

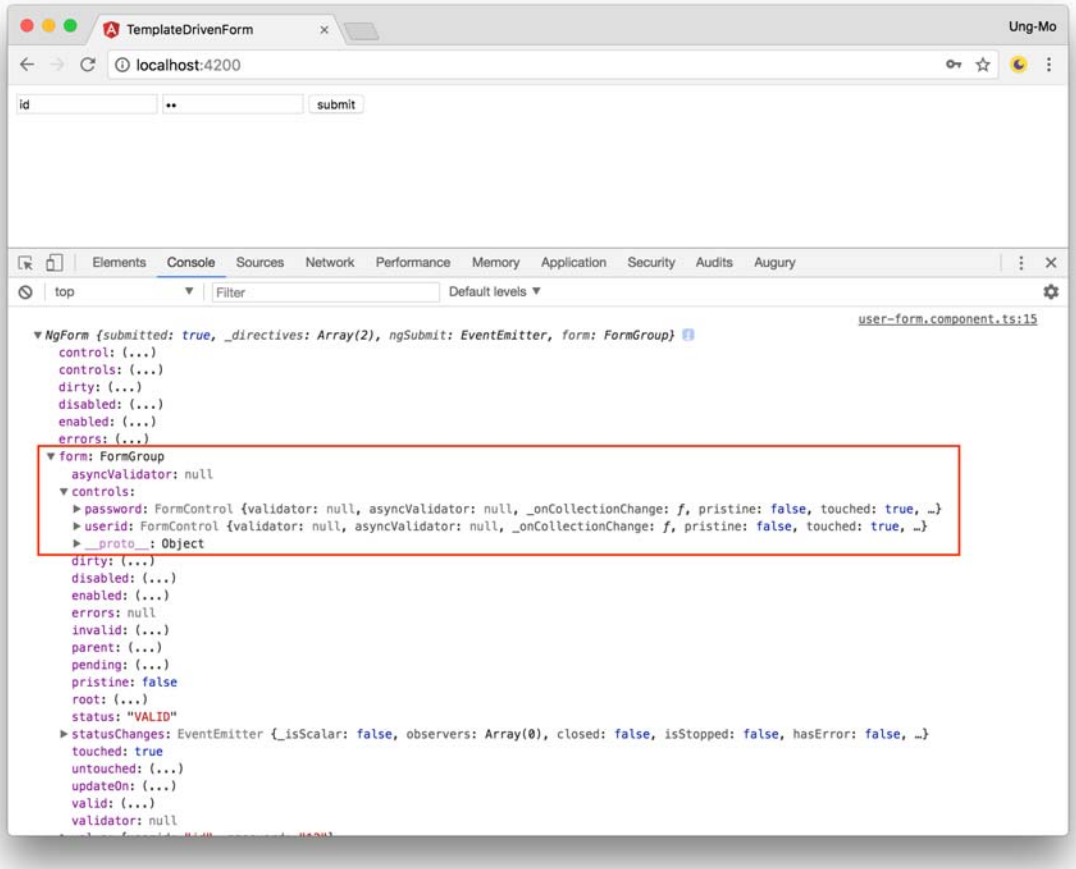
폼 요소의 자식 폼 컨트롤 요소에 NgModel 디렉티브를 적용하여 FormGroup 인스턴스에 의해 자식 폼 컨트롤 요소가 관리되도록 수정해 보자.

TYPESCRIPT

```
// user-form.component.ts
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'user-form',
  template: `
    <form #userForm="ngForm" (ngSubmit)="onNgSubmit(userForm)" novalidate>
      <input type="text" name="userid" placeholder="userid" ngModel>
      <input type="password" name="password" placeholder="password" ngModel>
      <button>submit</button>
    </form>
  `
})
export class UserFormComponent {
  onNgSubmit(userForm: NgForm) {
    console.log(userForm);
  }
}
```

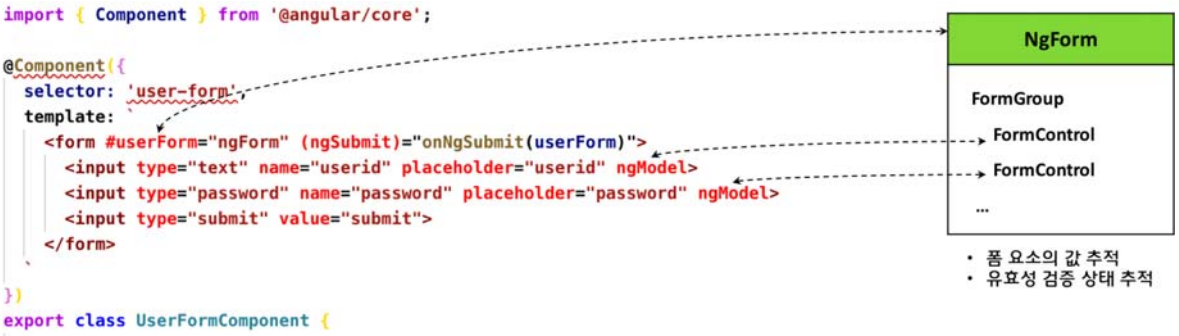
위 컴포넌트를 실행하여 보면 아래와 같이 NgModel 디렉티브가 적용된 자식 폼 컨트롤 요소를 가리키는 FormControl 인스턴스가 FormGroup 인스턴스에 추가되어 있는 것을 확인할 수 있다.



FormControl 인스턴스가 FormGroup 인스턴스에 추가되었다.

2.2 NgModel 디렉티브

NgModel 디렉티브는 자신이 적용된 폼 컨트롤 요소의 값이나 유효성 검증 상태의 추적 기능을 제공하는 FormControl 인스턴스를 생성한다고 하였다. 이 FormControl 인스턴스는 템플릿 기반 폼을 구성하는 기본 단위로서 폼 컨트롤 요소의 값이나 유효성 검증 상태를 추적하고 뷰와 폼 모델을 동기화된 상태로 유지한다.



NgForm 인스턴스 내에 생성된 FormControl

폼 컨트롤 요소의 값은 NgForm을 가리키는 템플릿 참조 변수 userForm의 value 프로퍼티 (userForm.value)를 참조하면 아래와 같은 결과를 확인할 수 있다. 폼의 userid에 “myid”, password

에 “1234”를 입력한 경우이다.

JSON

```
{
  "userid": "myid",
  "password": "1234"
}
```

이것은 폼 요소의 자식 요소 중 NgModel 디렉티브가 적용된 모든 자식 폼 컨트롤 요소의 상태 값을 나타낸다. 이때 키는 폼 컨트롤 요소의 name 어트리뷰트의 값이고, 값은 사용자 입력 값을 나타낸다. 따라서 **폼 컨트롤 요소에는 반드시 name 어트리뷰트를 지정하여야 한다.**

NgModel 디렉티브는 양방향 데이터 바인딩에서 사용할 때와는 달리 괄호와 할당문없이 선언한다.

HTML

```
<form #userForm="ngForm" novalidate>
  <input type="text" name="userid" ngModel>
  ...
```

폼 요소에 템플릿 참조 변수를 선언한 것과 같이 폼 콘트롤 요소에도 템플릿 참조 변수를 사용할 수 있다.

HTML

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

이때 템플릿 참조 변수 userid는 네이티브 DOM을 가리킨다. 템플릿 참조 변수 userid에 ngModel을 할당하면 userid는 네이티브 DOM을 가리키지 않고 userid 폼 컨트롤 요소를 가리키는 NgModel 인스턴스를 가리킨다.

NgModel 디렉티브는 자신이 적용된 폼 컨트롤 요소의 값이나 유효성 검증 상태의 추적 기능을 제공하는 FormControl 인스턴스를 생성한다고 하였다. NgModel 디렉티브는 FormControl 인스턴스뿐만 아니라 NgModel 인스턴스도 생성한다. 템플릿 참조 변수에 ngModel을 할당했을 때 반환되는 NgModel 인스턴스는 FormControl 인스턴스의 참조를 가지고 있으며 NgForm 인스턴스의 _directives 프로퍼티에 포함된다.

NgForm 인스턴스

사실 프레임워크가 내부적으로 관리하는 객체들에 대한 모든 것을 알 필요는 없다. NgModel 디렉티브는 자신이 적용된 폼 컨트롤 요소의 값이나 유효성 검증 상태의 추적 기능을 제공한다는 것이 중요하며 그 기능을 어떻게 사용하는지 아는 것이 중요하다. 이 장에서 설명하는 인스턴스들에 대해서는 참고 레벨로 알아보도록 하자.

따라서 **참조 변수에 ngModel을 할당하면 참조 변수를 통해 값 또는 유효성 검증 상태 추적이 가능해진다.**

HTML

```
<input type="text" name="userid" ngModel #userid="ngModel">

<!-- 템플릿 참조 변수를 통해 폼 컨트롤 요소의 값을 참조 -->
<p>userid value: {{ userid.value }}</p>
<!-- 템플릿 참조 변수를 통해 폼 컨트롤 요소의 유효성 검증 상태를 참조 -->
<p>userid valid: {{ userid.valid }}</p>
```

템플릿 참조 변수를 이벤트 핸들러에 인자로 전달하여 필요한 로직을 실행할 수 있다. 하지만 컴포넌트 클래스의 핸들러 함수가 필요한 로직을 실행한 이후, 그 결과를 다시 템플릿으로 보내야 한다면 양방향 데이터 바인딩을 사용하는 것이 보다 간편하다.

HTML

```
<input type="text" name="userid" [(ngModel)]="user.id" #userid="ngModel">
```

2.3 NgModelGroup 디렉티브

NgModelGroup 디렉티브는 NgForm 디렉티브와 유사하게 FormGroup 인스턴스를 생성하고 NgModelGroup 디렉티브가 적용된 폼 그룹 요소의 자식 요소 중에서 NgModel 디렉티브가 적용된 요소를 탐색하여 FormGroup 인스턴스에 추가한다.

아래 예제를 살펴보자.

TYPESCRIPT

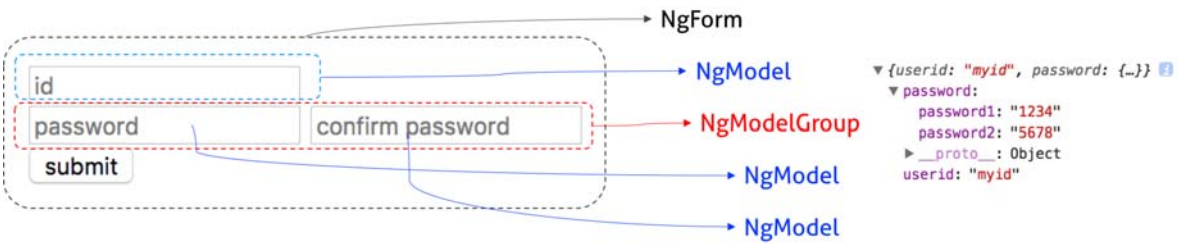
```
// user-form.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'user-form',
  template: `
    <form #userForm="ngForm" (ngSubmit)="onNgSubmit(userForm.value)" novalid
ate>
      <input type="text" name="userid" placeholder="id" ngModel>
      <div ngModelGroup="password">
        <input type="password" name="password1" placeholder="password" ngMod
el>
        <input type="password" name="password2" placeholder="confirm passwor
d" ngModel>
      </div>
      <button>submit</button>
    </form>
  `
})
export class UserFormComponent {
  onNgSubmit(user) {
    console.log(user);
    /*
      { userid: "myid",
        password: { password1: "11", password2: "11" }
      }
    */

    // 패스워드와 확인 패스워드의 일치 여부 확인
    if (user.password.password1 !== user.password.password2) {
      console.log('패스워드가 일치하지 않습니다!');
    }
  }
}
```

template-driven-form-4
By ungmo2
Run Project

위 예제의 경우, 폼 요소는 FormGroup 인스턴스를 생성하고 자식 폼 컨트롤 요소 중에서 ngModel 디렉티브가 적용된 요소와 ngModelGroup 디렉티브가 적용된 요소를 추가한다. 이것을 그림으로 표현하면 아래와 같다.



ngModelGroup 디렉티브

3. NgModel과 양방향 바인딩

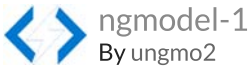
양방향 데이터 바인딩는 뷰와 컴포넌트 클래스의 상태 변화를 상호 반영하는 것을 말한다. 즉, 뷰의 상태가 변화하면 컴포넌트 클래스의 상태도 변화하고 그 반대로 컴포넌트 클래스의 상태가 변화하면 뷰의 상태도 변화하는 것이다.

NgModel 디렉티브는 앞에서 살펴본 바와 같이 폼 컨트롤 요소의 값과 유효성 검증 상태를 관리하는 FormControl 인스턴스를 생성한다고 하였다. 그런데 NgModel 디렉티브에 프로퍼티 바인딩을 사용하여 상태 프로퍼티를 바인딩하는 경우, 폼 컨트롤 요소의 상태 값을 업데이트할 수 있다.

TYPESCRIPT

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

@Component({
  selector: 'app-root',
  template: `<input [ngModel]="name">`
})
export class AppComponent {
  name = 'Lee';
}
```



그리고 NgModel 디렉티브에 `[()]` (이것을 Banana in a box라고 부른다) 문법을 사용하면 폼 컨트롤 요소의 상태 값과 컴포넌트 클래스의 상태 값을 동기화한다.

TYPESCRIPT

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

@Component({
  selector: 'app-root',
  template: `
    <input [(ngModel)]="name">
    <p>name: {{ name }}</p>
  `
})
export class AppComponent {
  name = 'Lee';
}
```

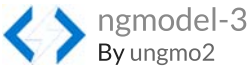
ngmodel-2
By ungmo2

사실 Angular는 양방향 바인딩을 지원하지 않는다. `[()]` 에서 추측할 수 있듯이 양방향 바인딩은 이벤트 바인딩과 프로퍼티 바인딩의 축약 표현(Shorthand syntax)일 뿐이다. 즉, 양방향 바인딩의 실제 동작은 이벤트 바인딩과 프로퍼티 바인딩의 조합으로 이루어진다. 위 코드를 이벤트 바인딩과 프로퍼티 바인딩으로 표현하여 보자.

TYPESCRIPT

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

@Component({
  selector: 'app-root',
  template: `
    <input [ngModel]="name" (ngModelChange)="name=$event">
    <p>name: {{ name }}</p>
  `
})
export class AppComponent {
  name = 'Lee';
}
```



이와 같이 양방향 바인딩은 ngModel 디렉티브와 ngModelChange 디렉티브 선언의 축약 표현으로 프로퍼티 바인딩과 이벤트 바인딩이 각각 처리된다. **ngModel 프로퍼티 바인딩은 컴포넌트 프로퍼티 name의 상태 변화를 수신하여 폼 컨트롤 요소의 상태를 업데이트하고 ngModelChange 이벤트 바인딩은 폼 컨트롤 요소의 상태 변화 이벤트를 방출하여 컴포넌트 프로퍼티 name의 상태를 업데이트한다.** 이때 ngModelChange은 $event로부터 상태 값(target.value)을 추출하여 name 프로퍼티에 할당한다.

양방향 바인딩은 반드시 NgModel 디렉티브를 사용하여야 하는 것은 아니며 커스텀 양방향 데이터 바인딩도 작성할 수 있다. 커스텀 양방향 바인딩의 간단한 예제를 작성해 보자. 먼저 부모 컴포넌트 역할을 담당할 루트 컴포넌트를 아래와 같이 작성한다.

TYPESCRIPT

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

@Component({
  selector: 'app-root',
  template: `
    <counter [(count)]="value"></counter>
    <p>Value: {{ value }}</p>
  `
})
export class AppComponent {
  value = 10;
}
```

앞서 작성한 AppComponent의 자식 컴포넌트 CounterComponent를 작성한다.

TYPESCRIPT

```
// counter.component.ts
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'counter',
  template: `
    <button (click)="decrement()">-</button>
    <button (click)="increment()">+</button>
  `
```

```
})
export class CounterComponent {
  @Input() count: number;
  @Output() countChange = new EventEmitter();

  decrement() {
    this.countChange.emit(--this.count);
  }

  increment() {
    this.countChange.emit(++this.count);
  }
}
```

부모 컴포넌트 AppComponent는 count 프로퍼티 바인딩을 통해 자식 컴포넌트에게 상태 정보를 전달한다. 자식 컴포넌트 CounterComponent는 @Input 데코레이터를 통해 입력 프로퍼티 count에 전달된 상태 정보를 바인딩한다. 또한 자식 컴포넌트 CounterComponent는 @Output 데코레이터와 함께 선언된 출력 프로퍼티 countChange를 EventEmitter 객체로 초기화한다. 그리고 부모 컴포넌트로 상태를 전달하기 위해 emit 메소드를 사용하여 이벤트를 발생시키면서 상태를 전달한다. 부모 컴포넌트는 자식 컴포넌트가 전달한 상태를 이벤트 바인딩을 통해 상태를 접수한다.

위 예제의 양방향 바인딩은 아래의 축약 표현으로 정확히 동일하게 동작한다.

HTML

```
<counter [count]="value" (countChange)="value=$event"></counter>
```

4. 템플릿 기반 폼 유효성 검증

NgForm, NgModel, NgModelGroup 디렉티브가 폼 컨트롤 요소에 적용되면 FormGroup 또는 FormControl 인스턴스를 생성한다. FormGroup과 FormControl는 유효성 검증 기능을 제공한다.

FormGroup와 FormControl는 AbstractControl를 상속한 클래스이다. AbstractControl 클래스는 valid, invalid, pristine, dirty, touched, untouched와 같이 요소의 유효성 검증 상태를 나타내는 프로퍼티를 소유하며 모든 자식 클래스에 상속한다. 이들 유효성 검증 상태 프로퍼티의 의미를 알아보자.

유효성 검증 상태 프로퍼티	의미

유효성 검증 상태 프로퍼티	의미
errors	유효성 검증에 실패한 경우, ValidationErrors 타입의 에러 객체를 반환한다. 유효성 검증에 성공한 경우, null를 반환한다.
valid	유효성 검증에 성공한 상태이면 true
invalid	유효성 검증에 실패한 상태이면 true
pristine	값을 한번도 입력하지 않은 상태이면 true
dirty	값을 한번 이상 입력한 상태이면 true
touched	focus in이 한번 이상 발생한 상태이면 true
untouched	focus in이 한번도 발생하지 않은 상태이면 true

아래의 예제를 살펴보자.

HTML

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

@Component({
  selector: 'app-root',
  template: `
    <input type="text"
      name="title"
      ngModel
      #title="ngModel"
      pattern="[a-zA-Z0-9]{4,10}"
      required>

    <p>errors:  {{ title.errors | json }}</p>
    <p>invalid: {{ title.invalid }}</p>
    <p>dirty:   {{ title.dirty }}</p>
    <p>touched: {{ title.touched }}</p>
    <p>pristine: {{ title.pristine }}</p>
  `
})
export class AppComponent {}
```

input 폼 컨트롤 요소에 대하여 required와 4자리 이상 10자리 이하의 영문 대소문자와 숫자만을 허용하는 pattern을 설정하였다. 이때 사용자가 pattern에 부합하는 값을 입력하면 valid는 true가 되고, pattern에 위배되는 값을 입력하면 invalid는 true가 된다. invalid가 true인 상태라면 errors에 에러의 내용을 담고 있는 ValidationErrors 타입의 에러 객체가 반환된다.

required가 설정되어 있으므로 값을 한 번도 입력하지 않은 상태, 즉 pristine이 true인 상태에도 invalid는 true이다. 이 경우는 사용자가 입력 필드에 값을 입력할 수도 없는 상태이므로 에러 메시지를 표시하지 않는 것이 좋다. 입력 필드에 포커스가 입력된 focus in 상태가 한 번 이상 발생한 경우, 즉 touched가 true인 상태라면 사용자가 입력 필드에 값의 입력을 시도하였다라고 판단해도 좋으므로 이때 에러의 유무를 체크하여 에러 메시지를 표시해도 좋다. 에러의 유무는 FormGroup 또는 FormControl 인스턴스의 error 프로퍼티의 값으로 판단할 수 있다. error 프로퍼티의 값이 null이면 유효성 검증에 성공한 것이고 error 프로퍼티의 값이 null이 아니면 유효성 검증에 실패한 것이다.

예를 들어 FormControl 인스턴스의 touched 프로퍼티가 true이고 FormControl 인스턴스의 errors?.required가 null이 아니라면 입력 필드에 포커스를 입력한 focus in 상태이나 값을 입력하지 않고 포커스를 입력 필드 바깥으로 이동시킨 focus out 상태이다. 이때 해당 입력 필드는 필수 입력 항목임을 사용자에게 알려 주기 위해 에러 메시지를 표시하여야 한다.

HTML

```
<input
  type="text"
  name="title"
  ngModel
  #title="ngModel"
  pattern="[a-zA-Z0-9]{4,10}"
  required>
<em *ngIf="title.errors?.required && title.touched">
  title을 입력하세요!
</em>
```

template-drive-form-validation-err-msg-1
By ungmo2

Run Project

만약 입력 필드에 focus in이 한 번 이상 발생한 상태, 즉 touched가 true이고 errors?.pattern이 null이 아니라면 유효성 검증 패턴을 통과하지 못한 상태이다. 이때 유효성 검증 패턴을 통과할 수 있는 값을 입력하도록 사용자에게 에러 메시지를 표시하여야 한다.

HTML

```
<input
  type="text"
  name="title"
  ngModel
  #title="ngModel"
  pattern="[a-zA-Z0-9]{4,10}"
  required>
<em *ngIf="title.errors?.pattern && title.touched">
  title은 영문 또는 숫자로 4자리 이상 10이하로 입력하세요!
</em>
```

#5. 템플릿 기반 폼 유효성 검증 실습

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

BASH

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

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

JSON

```
{
  ...
  "projects": {
    "my-project": {
      ...
      "architect": {
        "build": {
          ...
          "options": {
            ...
            "styles": [
              "node_modules/bootstrap/dist/css/bootstrap.min.css",
              "src/styles.css"
            ],
            "scripts": []
          },
  ...
}
```

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 {}
```

템플릿 기반 폼을 사용하기 위해 루트 모듈에 FormsModule을 추가한다.

TYPESCRIPT

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

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

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

이제 아래와 같이 회원 가입 폼 템플릿을 작성한다. 유효성 검증이 필요한 폼 컨트롤 요소에 required, pattern과 같은 빌트인 검증기(Built-in validator)를 선언한다.

HTML

```
<!-- user-form/user-form.component.html -->
<div class="container">
  <h2>Template-driven forms Exam</h2>
  <form #userForm="ngForm" (ngSubmit)="onSubmit(userForm)" novalidate>
    <div class="form-group">
      <label for="userid">User id</label>
      <input type="text" name="userid" class="form-control"
        [(ngModel)]="user.userid"
        #userid="ngModel"
        pattern="^[0-9a-zA-Z]([-_\.]?[0-9a-zA-Z])*@[0-9a-zA-Z]([-_\.]?[0-9a-
zA-Z])*\.[a-zA-Z]{2,3}$"
        required>
      <em *ngIf="userid.errors?.pattern && userid.touched" class="alert">
          User id는 email 형식으로 입력하세요!
      </em>
      <em *ngIf="userid.errors?.required && userid.touched" class="alert">
        User id로 사용할 email을 입력하세요!
      </em>
      <em>(touched: {{ userid.touched }} | pristine: {{ userid.pristine }} |
 invalid: {{ userid.invalid }})</em>
    </div>

    <div class="form-group">
      <label for="password">Password</label>
      <input type="password" name="password" class="form-control"
        [(ngModel)]="user.password"
        #password="ngModel"
        pattern="[a-zA-Z0-9]{4,10}"
        required>
      <em *ngIf="password.errors?.pattern && password.touched" class="alert"
>
        Password는 영문 또는 숫자로 4자리 이상 10이하로 입력하세요!
      </em>
      <em *ngIf="password.errors?.required && password.touched" class="aler
t">
        Password를 입력하세요!
      </em>
      <em>
        (touched: {{ password.touched }} | pristine: {{ password.pristine }}
 | invalid: {{ password.invalid }})
```

```
      </em>
    </div>

    <div class="form-group">
      <label for="role">Role</label>
      <select class="form-control" name="role"
        [(ngModel)]="user.role"
        required>
        <option *ngFor="let role of roles; let i=index;" [value]="role">
          {{ role }}
        </option>
      </select>
    </div>

    <div class="form-group">
      <label for="username">User name</label>
      <input type="text" name="name" class="form-control"
        [(ngModel)]="user.name">
    </div>

    <button type="submit" class="btn btn-success"
      [disabled]="userForm.invalid">Submit</button>
  </form>

  <pre>userForm.value: {{ userForm.value | json }}</pre>
  <pre>userForm.valid: {{ userForm.valid }}</pre>
  <pre>user: {{ user | json }}</pre>
</div>
```

모든 폼 컨트롤 요소가 유효성 검증에 성공한 상태(userForm.valid가 true 또는 userForm.invalid가 false)라면 submit 버튼이 활성화된다.

또한 NgForm, NgModel. NgModelGroup 디렉티브가 적용된 폼 컨트롤 요소에는 유효성 검증 상태 프로퍼티와 연동하여 ng-touched, ng-pristine, ng-invalid 등의 CSS 클래스가 자동 적용된다. 예를 들어 폼 컨트롤 요소의 유효성 검증 상태 프로퍼티 touched가 true이면 폼 컨트롤 요소에 ng-touched 클래스가 자동 적용되고, 유효성 검증 상태 프로퍼티 pristine이 true이면 폼 컨트롤 요소에 ng-pristine클래스가 자동 적용된다. 이들 CSS 클래스를 적절히 활용하면 유효성 검증 상태에 따른 스타일링이 가능하다.

컴포넌트 CSS는 아래와 같다.

CSS

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

input.ng-touched.ng-invalid {
  background-color: #ff6666;
}
```

컴포넌트 클래스는 아래와 같다.

TYPESCRIPT

```typescript
// user-form/user-form.component.ts
import { Component, OnInit } from '@angular/core';

class User {
  constructor(
    public userid: string,
    public password: string,
    public role: string,
    public name?: string
  ) {}
}

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

  ngOnInit() {
    this.roles = ['Admin', 'Developer', 'Guest'];
    this.initUser();
  }
```

```
    onSubmit(userForm) {
      console.log('Send user to server: ', this.user);
      userForm.reset();
    }

    initUser() {
      this.user = new User('', '', this.roles[0]);
    }
  }
```

템플릿 기반 폼은 작성이 간편하고 템플릿 내에서 유효성 검증 결과를 쉽게 확인 할 수 있는 장점이 있다. 하지만 폼이 커지면 마크업과 pattern 등의 유효성 검증 코드가 뒤섞여 복잡해 지고 가독성이 떨어질 수 있다.

또한 유효성 검증 로직에 중복이 발생할 우려가 커지고 세밀한 유효성 검증이 곤란할 수도 있다. 비교적 간단한 폼에는 템플릿 기반 폼이 유용하지만, 복잡한 폼에는 모델 기반 폼(리액티브 폼)을 사용하는 것이 효과적이다.

Reference

- NgForm
- FormGroup
- NgModel
- FormControl
- NgModelGroup
- AbstractControl