Forms

Angular forms

Angular supports two ways of dealing with forms:

- **Model driven forms**, where most is done in your component.
  - Also knows as new functional reactive way
  - Added advantage: this is unittestable
  - Resides in the `ReactiveFormsModule`
- **Template driven forms**, where most is done in the HTML.
  - Also known as the old AngularJS way
  - Resides in the `FormsModule`

A basic form

```
<form>
    <label for="inputPostcode">Postcode:</label>
    <input type="text" id="inputPostcode" required>
    <button>Save</button>
</form>
```

Model driven: An Angular form

In the component, build your form and handle submission

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

@Component({
    selector: 'my-app',
    templateUrl: 'app.component.html'
})
export class AppComponent {
    postalCodeForm: FormGroup;

    constructor() {
        this.postalCodeForm = new FormGroup({
            postalCode: new FormControl('') // initial value
        });
    }

    save() {
        // All inputs are accumulated onto the value
        console.log('Save postal code: ', this.postalCodeForm.value);
    }
}
```

Model driven: An Angular form

Then simply define your HTML with references to your component:

```
<form (ngSubmit)="save()" [formGroup]="postalCodeForm">
    <label for="inputPostalCode">Postal code:</label>
    <input type="text" id="inputPostalCode" formControlName="postalCode">
    <button>Save</button>
</form>
```

Model driven: An Angular form

And finally import the `ReactiveFormsModule` in your app module:

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

@NgModule({
    declarations: [...],
    imports: [
        //...
        ReactiveFormsModule
    ],
    providers: [...]
})
export class AppModule { }
```

Model driven: Validation

In the component, define your validations

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

@Component({
    selector: 'my-app',
    templateUrl: 'app.component.html'
})
export class AppComponent {
    postalCodeForm: FormGroup;

    constructor() {
        this.postalCodeForm = new FormGroup({
            postalCode: new FormControl('', [Validators.required])
        });
    }

    save() {
        // All inputs are accumulated onto the value
        console.log('Save postal code: ', this.postalCodeForm.value);
    }
}
```

Model driven: Validation

Make use of various validation metadata:

```
<form (ngSubmit)="save()" [formGroup]="postalCodeForm">
    <label for="inputPostalCode">Postal code:</label>
    <input type="text" id="inputPostalCode" formControlName="postalCode">

    <div *ngIf="postalCodeForm.get('postalCode').dirty &&
                postalCodeForm.get('postalCode').invalid">
        This is so invalid.
    </div>

    <button [disabled]="postalCodeForm.invalid">Save</button>
</form>
```

Model driven: Built-in validators

Angular comes with the following validators built-in:

- Validators.required
- Validators.pattern

```
constructor() {
    this.postalCodeForm = new FormGroup({
        postalCode: new FormControl('', [Validators.pattern('^[A-Z0-9]+$')])
    });
}
```

- Validators.minLength
- Validators.maxLength

Model driven: Shorter form group

With a lot of form controls, try this shorthand notation

```
import { Component } from '@angular/core';
import { FormGroup, Validators, FormBuilder } from '@angular/forms';

@Component({
    selector: 'my-app',
    templateUrl: 'app.component.html'
})
export class AppComponent {
    postalCodeForm: FormGroup;

    constructor(private fb: FormBuilder) {
        this.postalCodeForm = this.fb.group({
            postalCode: ['', [Validators.required]]
        });
    }

    save() {
        // All inputs are accumulated onto the value
        console.log('Save postal code: ', this.postalCodeForm.value);
    }
}
```

Model driven: Change detection

Get notified of changes:

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

@Component({
    selector: 'search',
    templateUrl: 'search.component.html'
}))
export class SearchComponent implements OnInit {
    searchControl = new FormControl();

    doSomething() {
        this.searchControl.valueChanges.subscribe(value => {
            // do something with value here
        });
    }
}
```

Template driven: An Angular form

In the HTML, use `ngModel`

```
<form (ngSubmit)="save()" novalidate>
    <label for="inputPostalCode">Postal code:</label>
    <input type="text"
           id="inputPostalCode"
           [(ngModel)]="postalCode"
           name="postalCode">

    <button>Save</button>
</form>
```

Template driven: An Angular form

In the component, only handle the submission of the form

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

@Component({
    selector: 'my-app',
    templateUrl: 'app.component.html'
})
export class AppComponent {
    postalCode: string;

    save() {
        console.log('Save postal code:', this.postalCode);
    }
}
```

Template driven: Form-level validation

Use a template reference variable to access the form metadata

```
<form (ngSubmit)="save()" novalidate #f="ngForm">
    <label for="inputPostalCode">Postal code:</label>
    <input type="text"
           id="inputPostalCode"
           [(ngModel)]="postalCode"
           name="postalCode">

    <button [disabled]="f.invalid">Save</button>
</form>
```

Template driven: Input level validation

Use a template reference variable to access the input metadata

```
<form (ngSubmit)="save()" novalidate>
    <label for="inputPostalCode">Postal code:</label>
    <input type="text"
           id="inputPostalCode"
           [(ngModel)]="postalCode"
           name="postalCode"
           #postalCode="ngModel"
           required>
    <div *ngIf="postalCode.dirty && postalCode.invalid">
        This is so invalid.
    </div>
    <div *ngIf="postalCode.errors?.required">
        This field is required.
    </div>

    <button>Save</button>
</form>
```

Built-in validators

Angular comes with the following validators built-in:

- required

```
<input [(ngModel)]="postalCode" required>
```

- pattern

```
<input [(ngModel)]="postalCode" pattern="^[a-zA-Z]{4}[0-9]{2}$">
```

- minlength/maxlength

```
<input [(ngModel)]="postalCode" minlength="6" maxlength="7">
```

Form validation

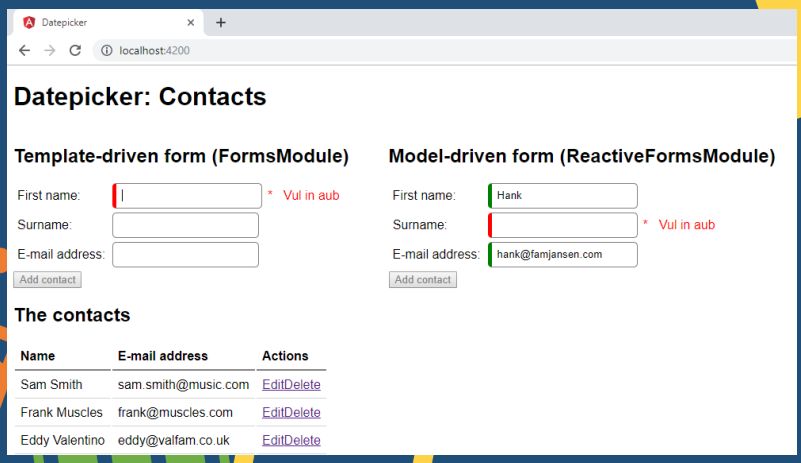
Angular places CSS classes on input fields based on their state.

- `.ng-valid`: value is valid
- `.ng-invalid`: value is not valid
- `.ng-pristine`: value has not been changed yet
- `.ng-dirty`: value has been changed
- `.ng-touched`: blur event has not occurred
- `.ng-untouched`: blur event has occurred

Recap

- Two ways of working with forms:
  - Template driven, where most is done in the HTML
  - Model driven, where most is done from component
    - More dynamic
    - Thus, unit testable
- Handling a form submit
- Using CSS with validation
- Showing validation messages
- Apply generic form validation by disabling buttons

Lab: form validation



Dynamic input fields

Input fields that are dynamic are a bit special in Angular.

```
<form [formGroup]="candyForm">
    <p>What candies do you like most?</p>

    Your name: <input formControlName="name">
    <ul>
        <li *ngFor="let c of candies">
            <input type="checkbox" formControlName="...??">
            {{c.name}}
            why:
            <input type="text" formControlName="...??">
        </li>
    </ul>

    <pre>{{candyForm.value | json}}</pre>
</form>
```

Dynamic input fields

What we want, is values to be bound like this:

What candies do you like most?

Your name: Frank

- ☐ Twix why:
- ☑ Skittles why: love the colors
- ☐ Twinkies why:
- ☑ Smarties why: still the colors


```
{
  "name": "Frank",
  "candies": [
    {
      "like": false,
      "name": "Twix",
      "why": ""
    },
    {
      "like": true,
      "name": "Skittles",
      "why": "love the colors"
    },
    {
      "like": false,
      "name": "Twinkies",
      "why": ""
    },
    {
      "like": true,
      "name": "Smarties",
      "why": "still the colors"
    }
  ]
}
```


Dynamic input fields

For these scenarios, `formArrayName` is designed to help you bind to the corrected values.

```
<form [formGroup]="candyForm">
    <p>What candies do you like most?</p>

    Your name: <input formControlName="name">
    <ul formArrayName="candies">
      <li *ngFor="let c of candies; let i = index" [formGroupName]="i">
        <input type="checkbox" formControlName="like">
              {{c.name}}
              why:
              <input type="text" formControlName="why">
      </li>
    </ul>

    <pre>{{candyForm.value | json}}</pre>
</form>
```

Dynamic input fields

Because you're databinding to specific indexes in the array, those items need to exist. Every candy needs to have a representation in the form definition.

```
this.candies = [
    { name: 'Twix' }, { name: 'Skittles' },
    { name: 'Twinkies' }, { name: 'Smarties' }
];

let formCandies: FormGroup[] = [];
for (let c of this.candies) {
    formCandies.push(this.fb.group({
        like: false,
        name: c.name,
        why: ''
    }));
}

this.candyForm = this.fb.group({
    name: [''],
    candies: this.fb.array(formCandies)
});
```