Lab: Forms

Before doing anything, we need to make sure the form features of Angular are activated.

In `src/app/app.module.ts`, make sure the `FormsModule` is imported:

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

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

This `FormsModule` contains the `ngModel` attribute and with this module, Angular actually performs form validation.

Exercise 1: Styling

1. In `src/styles.css`, add some CSS to give your form fields some color based on their state

   ```
   .ng-valid {
        /* some coloring here */
   }
   .ng-invalid {
        /* some coloring here */
   }
   ```

2. In `app.component.html`, add validation to your form. `required`, `pattern`, `maxlength` and/or `minlength`

3. Your form now immediately shows these stylings as soon as the page is loaded. Play around with CSS classes `.ng-touched`, `.ng-untouched`, `.ng-pristine` and `.ng-dirty` to refine this behavior.

Exercise 2: Disable the button on invalid

1. Add a template reference variable on your form

   ```
   <form (ngSubmit)="addContact()" #addContactForm="ngForm">
   ```

2. Bind the `disabled` property to `addContactForm.valid`

   ```
   <button class="btn btn-primary"
           [disabled]="!addContactForm.valid">Add</button>
   ```

   You might need to tweak your CSS to visibly reflect that the button has actually been disabled.

Exercise 3: Validation messages

1. In `app.component.html`, place an `ngModel` with a template reference variable on the input fields to make the metadata accessible.

```
<input type="text"
       required
       pattern=".+@.+\.nl"
       name="email"
       id="inputEmail"
       class="form-control"
       [(ngModel)]="newContact.email"
       #email="ngModel">
```

2. Create multiple `div`s with an `*ngIf` on it that checks whether a certain validation rule has been violated.

```
<div *ngIf="email.errors?.required">Please enter your e-mail
address</div>
```

Exercise 4: Model-driven approach

We will now try the model-driven approach. At the end of this exercise, our form should work just as it does now.

1. In `src/app/app.module.ts`, replace the `FormsModule` by the `ReactiveFormsModule`:

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

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

2. In `src/app/app.component.html`, bring the form back to its basics by removing all Angular forms attributes (`ngModel` and template reference variables) as well as the validation attributes (`required`, `pattern`, etc).

3. In `app.component.ts`, import the form types

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

4. Add a field representing the form

```
export class AppComponent {
    addContactForm: FormGroup;
```

5. Dependency inject a `FormBuilder` and use it to build the form:

```
constructor(private fb: FormBuilder) {
    this.addContactForm = this.fb.group({
        // all fields
        email: ['', Validators.pattern('^.+@.+\.nl
)]
    });
}
```

6. In `src/app/app.component.html`, add a `[formGroup]` reference to the FormGroup:

```
<form (ngSubmit)="addContact()" [formGroup]="addContactForm">
```

7. And register the different controls:

```
<input type="email" formControlName="email">
```

8. To retrieve the values posted in the form, use `.value` on the FormGroup:

```
addContact() {
    console.log('Submitted value:', this.addContactForm.value);
}
```

Challenge! A custom validator

You can create your own validators. Your challenge: Create a `emailValidator` validator.

```
this.addContactForm = this.fb.group({
    // all fields
    email: ['', emailValidator]
});
```