1. Atelier - Formulaires basés sur des modèles

Développer des formulaires nécessite des compétences de conception, ainsi qu'un support de base pour la liaison de données bidirectionnelle, le suivi des modifications, la validation et la gestion des erreurs. C'est ce qu'on va développer dans ce TP.

I. Objectifs

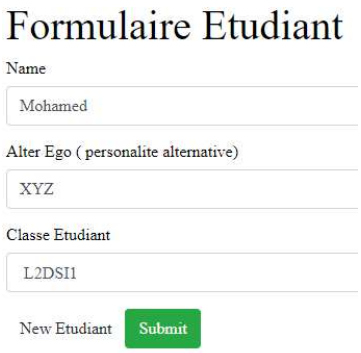
Ce TP montre comment créer un formulaire simple. Vous apprendrez comment :

- Construire un formulaire Angular avec un composant et un modèle.
- Créer des liaisons de données bidirectionnelles pour la lecture et l'écriture de valeurs de contrôle d'entrée.
- Suivre les changements d'état et la validité des contrôles de formulaire.
- Fournissez un retour visuel à l'aide de classes CSS spéciales qui suivent l'état des contrôles.
- Afficher les erreurs de validation aux utilisateurs et activer/désactiver les contrôles de formulaire.
- Partagez des informations sur des éléments HTML à l'aide de variables de référence de modèle.

On peut créer des formulaires en écrivant des modèles à l'aide de la syntaxe Modèle Angular des directives et techniques spécifiques aux formulaires décrites dans cette page.

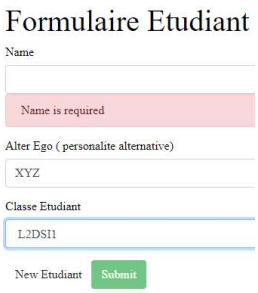
Vous pouvez également utiliser une approche réactive (ou basée sur un modèle) pour créer des formulaires. Cependant, ce TP se concentre sur les formulaires basés sur des modèles.

Angular facilite le processus en effectuant de nombreuses tâches répétitives. On se propose de créer un formulaire basé sur un modèle qui ressemble à ceci :



Deux des trois champs de ce formulaire sont obligatoires. Les champs obligatoires comportent une barre verte sur la gauche pour les rendre facilement repérables. Si vous

supprimez le nom, le formulaire affiche une erreur de validation dans un style qui attire l'attention :



Notez que le bouton "Envoyer" est désactivé et que la barre "obligatoire" située à gauche du contrôle d'entrée passe du vert au rouge. Vous pouvez personnaliser les couleurs et l'emplacement de la barre "obligatoire" avec du CSS standard. Vous allez construire ce formulaire par petites étapes :

- Créez le modèle `ClasseEtudiant`.
- Créez le composant qui contrôle le formulaire.
- Créez un modèle avec la disposition initiale du formulaire.
- Liez les propriétés de données à chaque contrôle de formulaire à l'aide de la syntaxe `ngModel` de liaison de données bidirectionnelle.
- Ajoutez un attribut `name` à chaque contrôle de saisie de formulaire.
- Ajoutez des `CSS` personnalisés pour fournir un retour visuel.
- Afficher et masquer les messages d'erreur de validation.
- Gérez la soumission de formulaire avec `ngSubmit`.
- Désactivez le bouton "Envoyer" du formulaire jusqu'à ce que le formulaire soit valide.

II. Créer un composant de formulaire

1) Créez un nouveau projet nommé `TP1-forms` :

```
ng new TP3-forms
```

puis

```
cd .\TP3-forms\
```

Un modèle peut être aussi simple qu'un ensemble de propriétés contenant des informations importantes sur une application. Voilà qui décrit bien la `ClasseEtudiant` avec ses trois champs obligatoires (`id`, `name`, `class`) et un champ facultatif (`Surname`).

2) A l'aide de la commande CLI, générez une nouvelle classe nommée `Etudiant` :

```
ng generate class Etudiant
```

Avec ce contenu :

```
export class Etudiant {
  constructor(
    public id: number,
    public name: string,
    public classe: string,
    public Surname?: string
  ) { }
}
```

Le compilateur `TypeScript` génère un champ `public` pour chaque paramètre public du `constructor` et affecte automatiquement la valeur du paramètre à ce champ lorsque vous créez un `Etudiant`.

Le `Surname` est optionnel (notez le point d'interrogation `?`) ceci permet au `constructor` de l'omettre :

3) Un formulaire Angular comprend deux parties : un modèle `HTML` et une classe de composants pour gérer les interactions de données et utilisateur par programme. Commencez par le cours, car il indique brièvement ce que l'éditeur de étudiants peut faire.

A l'aide de la commande CLI générez un nouveau composant nommé `EtudiantForm` :

```
ng generate component EtudiantForm
```

Avec ce contenu :

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

@Component({
  selector: 'app-etudiant-form',
  templateUrl: './etudiant-form.html',
  styleUrls: ['./etudiant-form.css']
})
export class EtudiantFormComponent {

  classes = ['L2DSI1', 'L2DSI2',
             'L2DSI3', 'L3DSI1','L3DSI2'];

  model = new Etudiant(18, 'Mohamed', this.classes[0], 'XYZ');
  submitted = false;
  onSubmit() { this.submitted = true; }

  // TODO: Remove this when we're done
```

```
 get diagnostic() { return JSON.stringify(this.model); }
  newEtudiant() {
    this.model = new Etudiant(42, '', '');
  }
}
```

Une petite explication :

- Le code importe la bibliothèque de noyau Angular et la classe `Etudiant` que vous venez de créer.
- La valeur "`app-etudiant-form`" du sélecteur `@Component` signifie que vous pouvez déposer ce formulaire dans un modèle parent avec une balise `<app-etudiant-form>`.
- La propriété `templateUrl` pointe sur un fichier séparé pour le modèle `HTML`.
- Vous avez défini des données factices pour `model` et `classes` comme il convient pour une démonstration.
- La propriété `diagnostic` pour renvoyer une représentation `JSON` du modèle. Cela nous aidera à voir ce qu'on fait pendant le développement. Cette partie doit être effacée une fois le TP terminé.

4) `app.module.ts` définit le module racine de l'application. On y identifie les modules externes qu'on utilise dans l'application et déclare les composants qui appartiennent à ce module, tels que le `EtudiantFormComponent`.

Les formulaires basés sur des modèles étant dans leur propre module, vous devez ajouter le `FormsModule` à la partie des `imports` pour le module d'application avant de pouvoir utiliser des formulaires.

Donc, mettez `app.module.ts` à jour avec les éléments suivants :

- Importez `FormsModule`.
- Ajoutez `FormsModule` à la liste de imports définie dans le décorateur `@NgModule`. Cela donne à l'application un accès à toutes les fonctionnalités des formulaires basés sur des modèles, y compris `ngModel`.

Si un composant, `directive` ou `pipe` appartient à un module dans le tableau `imports`, ne pas le re-déclarer dans le tableau `déclarations`. Si vous l'avez écrit et il devrait appartenir à ce module, ne le déclarer pas dans le tableau de déclarations.

Le code source est alors comme suit :

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

```
import { AppComponent }  from './app';
import { EtudiantFormComponent } from './etudiant-form/etudiant-form';

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

5) `AppComponent` est le composant racine de l'application. Il va accueillir le nouveau `EtudiantFormComponent`.

Remplacez le contenu de `app.html` par ce qui suit:

```
<app-etudiant-form></app-etudiant-form>
```

Il n'y a que deux changements. La template est simplement la nouvelle balise d'élément identifiée par la propriété `selector` du composant. Cela affiche le formulaire de l'étudiant lorsque le composant d'application est chargé. N'oubliez pas de supprimer également le champ `name` du corps de la classe.

III. Créer un modèle de formulaire HTML initial

1) Mettez à jour le fichier `etudiant-form.html` avec le contenu suivant :

```
<div class="container">
    <h1>Formulaire Etudiant</h1>
    <form>
      <div class="form-group">
        <label for="name">Name</label>
        <input type="text" class="form-control" id="name" required>
      </div>
      <div class="form-group">
        <label for="Surname">SurName</label>
        <input type="text" class="form-control" id="Surname">
      </div>
      <button type="submit" class="btn btn-success">Submit</button>
    </form>
```

```
</div>
```

Vous présentez deux des champs `Etudiant, name` et `Surname`, et les ouvrir pour l'entrée de l'utilisateur dans des boîtes d'entrée. Le contrôle `<input> Name` a l'attribut `required` en HTML5. Le contrôle `<input> Surname` ne le fait pas car `Surname` est optionnel. Vous avez ajouté un bouton `submit` en bas avec quelques classes pour le `style`. Vous n'utilisez pas encore Angular. Il n'y a pas de liaisons ou de directives supplémentaires, juste une mise en page. Dans les formulaires basés sur des modèles, si vous avez importé `FormsModule`, vous n'avez rien à faire avec la balise <form> pour pouvoir l'utiliser `FormsModule`. Continuez pour voir comment cela fonctionne. Angular ne fait aucune utilisation des container, `form-group, form-control` et les `btnclasses` ou les `styles` d'une bibliothèque externe. Les applications Angulare peuvent utiliser n'importe quelle bibliothèque CSS ou aucune.

2) Pour ajouter la feuille de style, ouvrez `styles.css` et remplacer la par la feuille du style `Bootstrap v4.5.3` (à télécharger à partir du net) et faite extraire le fichier `Bootstrap.css`.
3) Vous allez ajouter un `select` au formulaire et lier les options à la liste `classes` en utilisant `ngFor`.

Ajoutez le code HTML suivant immédiatement sous le groupe `Surname` :

```
      <div class="form-group">
            <label for="classe">Classe</label>
            <select class="form-control" id="classe" required>
              <option *ngFor="let pow of classes" [value]="pow">{{pow}}</option>
            </select>
        </div>
```

Remplacer ***ngFor*** par ***@for***

Ce code répète l'étiquette `<option>` pour chaque pouvoir de la liste des pouvoirs. La variable `pow` d'entrée modèle est une puissance différente à chaque itération; vous affichez son `nom` en utilisant la syntaxe d'interpolation.

1) Liaison de données bidirectionnelle avec `ngModel`

Lancer l'application maintenant serait décevant.

Vous devez maintenant afficher, écouter et extraire en même temps. Vous pouvez utiliser les techniques que vous connaissez déjà, mais vous utiliserez plutôt la nouvelle syntaxe `[(ngModel)]`, ce qui facilite la liaison du formulaire au modèle.

Recherchez la balise `<input>` de `name` et mettez-la à jour comme ceci :

```
  <input type="text" class="form-control" id="name" required
                [(ngModel)]="model.name" name="name">
                TODO: remove this: {{model.name}}
```

Vous avez ajouté une interpolation de diagnostic après la balise d'entrée pour que vous puissiez voir ce que vous faites.

Mettre l'accent sur la syntaxe de liaison: `[(ngModel)]="..."` Vous avez besoin d'un autre ajout pour afficher les données. Déclarez une variable de modèle pour le formulaire. Mettez à jour la `<form>` balise avec comme suit : `#etudiantForm="ngForm"`

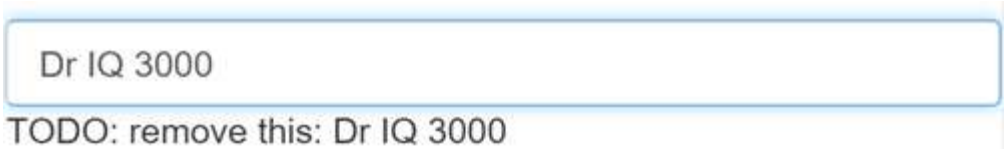
```
<form #etudiantForm="ngForm" >
```

La variable `etudiantForm` est maintenant une référence à la directive `NgForm` qui régit le formulaire dans son ensemble.

IV. La directive NgForm

La directive `NgForm` complète l'élément `form` avec des fonctionnalités supplémentaires. Il contient les contrôles que vous avez créés pour les éléments avec une directive `ngModel` et un attribut name et surveille leurs propriétés, y compris leur validité. Il possède également sa propre propriété `valid`, qui n'est vraie que si chaque contrôle contenu est valide.

Si vous exécutez l'application maintenant et commencez à taper dans la zone de saisie `Name`, ajouter et à supprimer des caractères, vous les verrez apparaître et disparaître du texte interpolé, à un moment donné, cela pourrait ressembler à ceci :



Le diagnostic est la preuve que les valeurs découlent réellement de la zone de saisie vers le modèle et inversement. C'est une liaison de données bidirectionnelle.

Notez que vous avez également ajouté un attribut `name` à la balise `<input>` et que vous l'avez défini sur "`name`", ce qui a du sens pour le nom du étudiant. Toute valeur unique fera l'affaire, mais utiliser un nom descriptif est utile. Définir un attribut `name` est une exigence lorsqu'on l'utilise en combinaison avec un formulaire `[(ngModel)]`

En interne, Angular crée des instances `FormControl` et les enregistre avec une directive `NgForm` attachée par Angular à la balise `<form>`. Chaque `FormControl` est enregistré sous le nom que vous avez attribué à l'attribut `name`.

Ajoutez des liaisons `[(ngModel)]` et des attributs `name` similaires à `Surname` et à `Classe`. Vous allez abandonner le message de liaison de la zone de saisie et ajouter une nouvelle liaison (en haut) à la propriété `diagnostic` du composant. Vous pouvez ensuite confirmer que la liaison de données bidirectionnelle fonctionne pour le modèle de étudiants entier.

Après révision, le noyau du formulaire devrait ressembler à ceci:

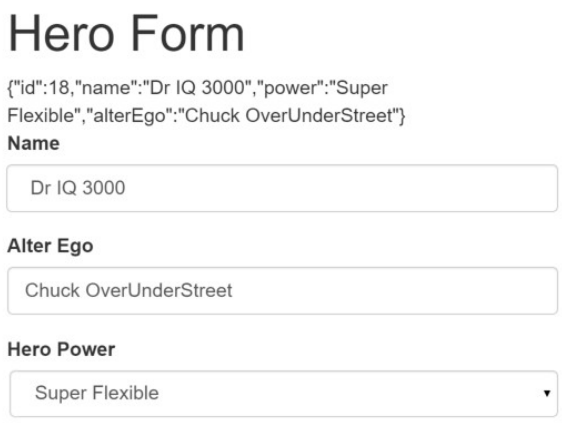
```
{{diagnostic}}
<div class="form-group">
  <label for="name">Name</label>
  <input type="text" class="form-control" id="name"
         required
         [(ngModel)]="model.name" name="name">
</div>

<div class="form-group">
  <label for="Surname">surname </label>
  <input type="text"  class="form-control" id="Surname"
         [(ngModel)]="model.Surname" name="Surname">
</div>

<div class="form-group">
  <label for="classe"> Classe de l'Etudiant  </label>
  <select class="form-control"  id="classe"
          required
          [(ngModel)]="model.classe" name="classe">
    <option *ngFor="let pow of classes" [value]="pow">{{pow}}</option>
  </select>
</div>
```

- Chaque élément d'entrée a une propriété `id` utilisée par l'attribut de l'élément `label` pour faire correspondre l'étiquette à son contrôle d'entrée.
- Chaque élément d'entrée possède une propriété `name` requise par les formulaires Angular pour enregistrer le contrôle avec le formulaire.

Si vous exécutez l'application maintenant et modifiez chaque propriété de modèle de étudiants, le formulaire peut s'afficher comme suit :



Le diagnostic situé en haut du formulaire confirme que toutes vos modifications sont reflétées dans le modèle. Supprimez l'interpolation {{ diagnostic }} en haut car elle a atteint son objectif.

V. Suivi de l'état et de la validité avec ngModel

L'utilisation `ngModel` dans un formulaire vous offre plus qu'une simple liaison de données bidirectionnelle. Il vous indique également si l'utilisateur a touché le contrôle, si la valeur a changé ou si la valeur est devenue invalide.

La directive `NgModel` ne fait pas que suivre l'état; il met à jour le contrôle avec des classes `CSS` Angular spéciales qui reflètent l'état. Vous pouvez utiliser ces noms de classe pour modifier l'apparence du contrôle.

Etat	Classe si vrai	Classe si faux
Le contrôle a été visité.	`ng-touched`	`ng-untouched`
La valeur du contrôle a changé.	`ng-dirty`	`ng-pristine`
La valeur du contrôle est valide.	`ng-valid`	`ng-invalid`

Ajoutez temporairement une variable de référence de modèle nommée `spy` à la balise `name` de `<input>` et utilisez-la pour afficher les classes `CSS` de l'entrée.

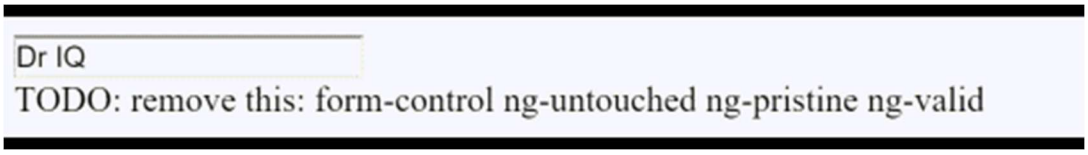
```
<input type="text" class="form-control" id="name"

        required
        [(ngModel)]="model.name" name="name"
        #spy>
        <br>TODO: remove this: {{spy.className}}
```

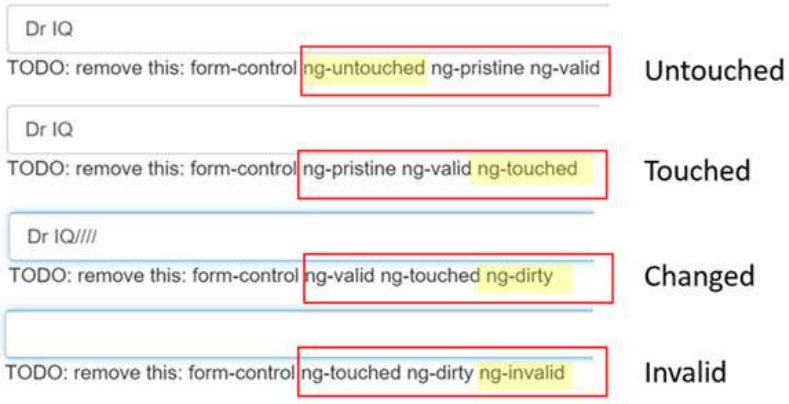
Maintenant, lancez l'application et regardez la zone de saisie `name`. Suivez ces étapes avec précision :

- Regarde mais ne touche pas.
- Cliquez dans la zone de `name`, puis cliquez en dehors de celle-ci.
- Ajouter des barres obliques à la fin du nom.
- Effacer le nom.

Les actions et effets sont les suivants :



Vous devriez voir les transitions et les noms de classe suivants:



La paire `ng-valid` / `ng-invalid` est la plus intéressante, car vous souhaitez envoyer un signal visuel fort lorsque les valeurs ne sont pas valides.

VI. Ajouter un CSS personnalisé pour les commentaires visuels

Vous pouvez marquer les champs obligatoires et les données non valides en même temps avec une barre de couleur à gauche du champ de saisie:

Vous obtenez cet effet en ajoutant ces définitions de classe à un nouveau fichier `forms.css` que vous ajoutez au projet en tant que frère de `index.html` :

Mettez à jour le `<head>` de `index.html` pour inclure cette feuille de style:

VII. Afficher et masquer les messages d'erreur de validation

Vous pouvez améliorer le formulaire. La zone de saisie `Name` est obligatoire et effacée, la barre devient rouge. Cela signifie que quelque chose ne va pas, mais que l'utilisateur ne sait pas ce qui ne va pas ou quoi faire. Utilisez l'état du contrôle pour révéler un message utile.

Lorsque l'utilisateur supprime le `name`, le formulaire doit ressembler à ceci:

Pour obtenir cet effet, étendez la balise `<input>` avec les éléments suivants :

- Une variable de référence de modèle .
- Le message " est requis " dans un `<div>`, que vous n'afficherez que si le contrôle est invalide.

Voici un exemple de message d'erreur ajouté à la zone de saisie du nom :

```
<label for="name">Name</label>
  <input type="text" class="form-control" id="name"
         required
         [(ngModel)]="model.name" name="name"
         #name="ngModel">
         <div [hidden]="name.valid || name.pristine"
              class="alert alert-danger">
           Name is required
         </div>
```

Vous avez besoin d'une variable de référence de modèle pour accéder au contrôle Angular de la zone de saisie à partir du modèle. Ici, vous avez créé une variable appelée `name` et lui avez donné la valeur `ngModel`.

Vous contrôlez la visibilité du message d'erreur de nom en liant les propriétés du contrôle `name` à la propriété de l'élément `<div>` de message `hidden`.

```
<div [hidden]="name.valid || name.pristine"
                class="alert alert-danger"> Name is required
        </div>
```

Dans cet exemple, vous masquez le message lorsque le contrôle est valide ou vierge; "vierge" signifie que l'utilisateur n'a pas changé la valeur depuis son affichage dans ce formulaire.

Cette expérience utilisateur est le choix du développeur. Certains développeurs souhaitent que le message soit affiché à tout moment. Si vous ignorez `pristine`, vous masquerez le message uniquement lorsque la valeur est valide. Si vous arrivez dans ce composant avec un nouveau `etudiant` (vierge) ou un `etudiant` invalide, le message d'erreur s'affiche immédiatement avant que vous n'ayez rien fait.

Certains développeurs souhaitent que le message ne s'affiche que lorsque l'utilisateur apporte une modification non valide. Cacher le message lorsque le contrôle est "vierge" permet d'atteindre cet objectif. Vous verrez la signification de ce choix lorsque vous ajoutez un nouveau `etudiant` au formulaire. `Surname` est facultatif, vous pouvez donc le laisser.

La sélection du pouvoir des étudiants est requise. Vous pouvez ajouter le même type de traitement des erreurs à la liste ‹select› si vous le souhaitez, mais ce n'est pas impératif, car la zone de sélection limite déjà le pouvoir à des valeurs valides.

Vous allez maintenant ajouter un nouveau `etudiants` sous cette forme. Placez un bouton "New Etudiant" au bas de la fiche et liez son événement `click` à une méthode `newEtudiant()` de composant.

```
<button type="button" class="btn btn-
default" (click)="newEtudiant()">New Etudiant</button>
```

Exécutez à nouveau l'application, cliquez sur le bouton "New Etudiant" et le formulaire s'efface. Les barres requises à gauche de la zone de saisie sont rouges, indiquant des propriétés non valides `name` et `classe`. C'est compréhensible car ce sont des champs obligatoires. Les messages d'erreur sont masqués car le formulaire est vierge.

Entrez un nom et cliquez à nouveau sur "New Etudiant". L'application affiche un message d'erreur Le nom est requis. Vous ne voulez pas de messages d'erreur lorsque vous créez un nouveau étudiant (vide). Pourquoi en obtenez-vous une maintenant ?

L'inspection de l'élément dans les outils du navigateur révèle que la zone de saisie du nom n'est plus vierge. Le formulaire rappelle que vous avez saisi un nom avant de cliquer sur "New Etudiant". Le remplacement de l'objet étudiant n'a pas restauré l'état initial des contrôles de formulaire.

Vous devez effacer impérativement tous les indicateurs, ce que vous pouvez faire en appelant la méthode `reset()` du formulaire après avoir appelé la méthode `newEtudiant()`.

Cliquez maintenant sur "Nouveau étudiants" pour réinitialiser le formulaire et ses indicateurs de contrôle.

VIII. Envoyez le formulaire avec ngSubmit

L'utilisateur doit pouvoir soumettre ce formulaire après l'avoir rempli. Le bouton "submit" situé au bas du formulaire ne fait rien, mais déclenche l'envoi d'un formulaire en raison de son type (`type="submit"`).

Une "soumission de formulaire" est inutile pour le moment. Pour le rendre utile, liez la propriété `ngSubmit` du formulaire à la méthode `onSubmit()` du composant de formulaire etudiant :

```
<form (ngSubmit)="onSubmit()" #etudiantForm="ngForm">
```

Vous aviez déjà défini une variable de référence de modèle `#etudiantForm` et l'aviez initialisée avec la valeur " `ngForm` ". Maintenant, utilisez cette variable pour accéder au formulaire avec le bouton "submit".

Vous lierez la validité globale du formulaire via la variable `etudiantForm` de la propriété `disabled` du bouton à l'aide d'une liaison d'événement. Voici le code :

```
<button type="submit" class="btn btn-
success" [disabled]="!etudiantForm.form.valid">Submit</button>
```

Si vous exécutez l'application maintenant, vous constaterez que le bouton est activé, bien qu'il ne fasse encore rien d'utile.

Maintenant, si vous supprimez le nom, vous ne respectez pas la règle "obligatoire", qui est dûment notée dans le message d'erreur. Le bouton "Soumettre" est également désactivé.

Pas impressionné ? Pensez-y un instant. Que feriez-vous pour relier l'état d'activation / désactivation du bouton à la validité du formulaire sans l'aide d'Angular ?

Pour vous, c'était aussi simple que cela:

- Définissez une variable de référence de modèle sur l'élément de formulaire (amélioré).
- Reportez-vous à cette variable dans un bouton à plusieurs lignes.

IX. Basculer entre deux régions de formulaire (crédit supplémentaire)

Pour un effet visuel plus frappant, masquez la zone de saisie des données et affichez autre chose. Envelopper le formulaire dans un `<div>` et lier sa propriété `hidden` à la propriété `EtudiantFormComponent.submitted`.

```
<div [hidden]="submitted">
  <h1>Etudiant Form</h1>
  <form (ngSubmit)="onSubmit()" #etudiantForm="ngForm">
     <!-- ... all of the form ... -->
  </form>
<div>
```

Le formulaire principal est visible depuis le début, car la propriété `submitted` est `false` jusqu'à ce que vous le soumettiez.

```
submitted = false;
onSubmit() { this.submitted = true; }
```

Lorsque vous cliquez sur le bouton "Submit", l'indicateur `submitted` devient `true` et le formulaire disparaît comme prévu.

L'application doit maintenant afficher quelque chose d'autre pendant que le formulaire est à l'état soumis. Ajoutez le code `HTML` suivant sous le `<div>` `wrapper` que vous venez d'écrire :

```
<div [hidden]="!submitted">
  <h2>You submitted the following:</h2>
  <div class="row">
    <div class="col-xs-3">Name</div>
    <div class="col-xs-9">{{ model.name }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Sur Name</div>
```

```
    <div class="col-xs-9">{{ model.Surname }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Classe</div>
    <div class="col-xs-9">{{ model.classe }}</div>
  </div>
  <br>
  <button class="btn btn-primary" (click)="submitted=false">Edit</button>
</div>
```

Il y a encore le etudiant, affiché en lecture seule avec des liaisons d'interpolation. Cela `<div>` n'apparaît que lorsque le composant est à l'état soumis. Le code `HTML` comprend un bouton "Modifier" dont l'événement `click` est lié à une expression qui supprime l'indicateur `submitted`. Lorsque vous cliquez sur le bouton "Modifier", ce bloc disparaît et le formulaire modifiable réapparaît.

X. Résumé

Le formulaire Angular décrite dans cette page tire parti des fonctionnalités suivantes de la structure pour prendre en charge la modification, la validation des données, etc… :

- Un modèle de formulaire `HTML` Angular.
- Une classe de composants de formulaire avec un décorateur `@Component`.
- Traitement de la soumission de formulaire en se liant à la propriété event `NgForm.ngSubmit`.
- Variables de référence de modèle telles que `#etudiantForm` et `#name`.
- `[(ngModel)]` syntaxe pour la liaison de données bidirectionnelle.
- Utilisation d'attributs `name` pour la validation et le suivi des modifications d'éléments de formulaire.
- Propriété valide de la variable de référence sur les contrôles d'entrée pour vérifier si un contrôle est valide et afficher / masquer les messages d'erreur.
- Contrôler l'état activé du bouton "submit" en le liant à la validité `NgForm`.
- Classes `CSS` personnalisées fournissant aux utilisateurs un retour visuel sur les contrôles non valides.