**12. Utiliser les événements et les formulaires**

Dans ce chapitre, je continue à décrire les fonctionnalités de base d'Angular, en me concentrant sur les caractéristiques qui répondent à **l'interaction** de l'utilisateur. J'explique comment créer des liaisons d'événements et comment utiliser des **liaisons bidirectionnelles** pour gérer le flux de données entre le **modèle** et le **gabarit(template)**. L'une des principales formes d'interaction avec l'utilisateur dans une application web est l'utilisation de formulaires HTML, et j'explique comment les liaisons d'événements et les liaisons bidirectionnelles de données sont utilisées pour les prendre en charge et valider le contenu fourni par l'utilisateur. Le tableau 12-1 replace les événements et les formulaires dans leur contexte.

Tableau 12-1Présentation des liaisons d'événements et des formulaires dans leur contexte

Question

Réponse

Qu'est-ce que c'est ?

Les liaisons d'événements évaluent une expression lorsqu'un événement est déclenché, par exemple lorsque l'utilisateur appuie sur une touche, déplace la souris ou soumet un formulaire. Les fonctionnalités plus larges liées aux formulaires s'appuient sur cette base pour créer des formulaires qui sont automatiquement validés afin de s'assurer que l'utilisateur fournit des données utiles.

Pourquoi sont-ils utiles ?

Ces fonctionnalités permettent à l'utilisateur de modifier l'état de l'application, en changeant ou en ajoutant des données dans le modèle.

Comment sont-ils utilisés ?

Chaque fonctionnalité est utilisée différemment. Voir les exemples pour plus de détails.

Y a-t-il des pièges ou des limites ?

Comme pour tous les bindings Angular, le principal piège est d'utiliser le mauvais type d'accolade pour indiquer un binding. Prêtez une attention particulière aux exemples de ce chapitre et vérifiez la façon dont vous avez appliqué les bindings lorsque vous n'obtenez pas les résultats escomptés.

Existe-t-il des alternatives ?

Non. Ces fonctionnalités font partie intégrante d'Angular.

Le tableau 12-2 résume le chapitre.

Tableau 12-2Résumé du chapitre

Problème

Solution

Liste

Activation de la prise en charge des formulaires

Ajouter le module @angular/forms à l'application

1-3

Réagir à un événement

Utiliser une liaison événementielle

4-6

Obtenir les détails d'un événement

Utiliser l'objet **$event**

7-9

Référence à des éléments du modèle

Définir les variables du modèle

10

Permettre le flux de données dans les deux sens entre l'élément et le composant

Utiliser une liaison de données bidirectionnelle

11, 12

Saisir les données de l'utilisateur

Utiliser un formulaire HTML

13, 14

Valider les données fournies par l'utilisateur

Effectuer la validation du formulaire

15-26

Préparation du projet d'exemple

Pour ce chapitre, je vais continuer à utiliser le projet d'exemple que j'ai créé au chapitre 9 et que j'ai modifié dans les chapitres suivants.

ConseilVous pouvez télécharger le projet d'exemple de ce chapitre et de tous les autres chapitres de ce livre à l'adresse https://github.com/Apress/pro-angular-5ed. Consultez le chapitre 1 pour savoir comment obtenir de l'aide si vous rencontrez des problèmes lors de l'exécution des exemples.

**Importation du module Forms**

Les fonctionnalités démontrées dans ce chapitre reposent sur le module Angular forms, qui doit être importé dans le module Angular, comme le montre le Listing 12-1.

import { NgModule } from '@angular/core' ;

import { BrowserModule } from '@angular/platform-browser' ;

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

import { BrowserAnimationsModule } from '@angular/platform-browser/animations' ;

import { ProductComponent } from './component' ;

import { FormsModule } from "@angular/forms" ;

@NgModule({

déclarations : [ProductComponent],

imports : [

BrowserModule,

BrowserAnimationsModule,

FormsModule

],

fournisseurs : [],

bootstrap : [ProductComponent]

})

export class AppModule { }

Liste 12-1Déclarer une dépendance dans le fichier app.module.ts du dossier src/app

La propriété imports du décorateur NgModule spécifie les dépendances de l'application. L'ajout de FormsModule à la liste des dépendances active les fonctionnalités du formulaire et les rend disponibles pour une utilisation dans l'ensemble de l'application.

**Préparation du composant et du modèle**

L'illustration 12-2 supprime le constructeur et certaines méthodes de la classe de composant et ajoute une nouvelle propriété, nommée selectedProduct.

import { Component } from "@angular/core" ;

import { Model } from "./repository.model" ;

import { Product } from "./product.model" ;

@Component({

selector : "app",

templateUrl : "template.html"

})

export class ProductComponent {

modèle : Model = new Model() ;

getProduct(key : number) : Product | undefined {

return this.model.getProduct(key) ;

}

getProducts() : Produit[] {

return this.model.getProducts() ;

}

selectedProduct : string | undefined ;

}

Listing 12-2Simplification du composant dans le fichier component.ts du dossier src/app

La liste 12-3 simplifie le modèle du composant, ne laissant qu'un tableau qui est rempli à l'aide de la directive ngFor.

<div class="p-2">

<table class="table table-sm table-bordered">

<tr><th></th><th>Nom</th><th>Catégorie</th><th>Prix</th></tr>

<tr \*ngFor="let item of getProducts() ; let i = index">

<td>{{i + 1}}</td>

<td>{{nom de l'article}}</td>

<td>{{article.catégorie}}</td> <td>{article.catégorie}</td>

<td>{{article.prix}}</td>

</tr>

</table>

</div>

Listing 12-3Simplification du modèle dans le fichier template.html du dossier src/app

Pour démarrer le serveur de développement, ouvrez une invite de commande, naviguez jusqu'au dossier example et exécutez la commande suivante :

ng serve

Ouvrez une nouvelle fenêtre de navigateur et naviguez jusqu'à http://localhost:4200 pour voir le tableau illustré à la figure 12-1.

Figure 12-1Exécution de l'application d'exemple

Utilisation de la liaison d'événements

La liaison d'événements est utilisée pour répondre aux événements envoyés par l'élément hôte. La liste 12-4 illustre la liaison d'événements, qui permet à un utilisateur d'interagir avec une application Angular.

<div class="p-2">

<div class="bg-info text-white p-2">

Produit sélectionné : {{selectedProduct ? ? '(None)'}}

</div>

<table class="table table-sm table-bordered">

<tr><th></th><th>Nom</th><th>Catégorie</th><th>Prix</th></tr>

<tr \*ngFor="let item of getProducts() ; let i = index">

<td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>

<td>{{nom de l'article}}</td>

<td>{{article.catégorie}}</td>

<td>{{article.prix}}</td>

</tr>

</table>

</div>

Listing 12-4Utilisation de la liaison d'événement dans le fichier template.html du dossier src/app

Lorsque vous enregistrez les modifications apportées au modèle, vous pouvez tester la liaison en déplaçant le pointeur de la souris sur la première colonne du tableau HTML, qui affiche une série de chiffres. Lorsque la souris se déplace d'une ligne à l'autre, le nom du produit affiché dans cette ligne apparaît en haut de la page, comme le montre la figure 12-2.

Figure 12-2Utilisation d'une liaison d'événements

Il s'agit d'un exemple simple, mais il montre la structure d'une liaison d'événement, qui est illustrée dans la figure 12-3.

Figure 12-3Anatomie d'une liaison d'événements

Une liaison d'événements comporte les quatre parties suivantes :

**L'élément hôte** est la source d'événements pour la liaison.

**Les crochets ronds** indiquent à Angular qu'il s'agit d'une liaison événementielle, qui est une forme de **liaison unidirectionnelle** dans laquelle les données circulent de l'élément vers le reste de l'application.

**L'événement** spécifie l'événement pour lequel la liaison est utilisée.

**L'expression** est évaluée lorsque l'événement est déclenché.

<**td** (mouseover)=”selectedItem=item.name”> item.id</td>

élément hôte

parenthèses

événement

expression

En examinant la liaison dans l'illustration 12-4, vous pouvez voir que l'élément hôte est un élément td, ce qui signifie qu'il s'agit de l'élément qui sera la source des événements. La liaison spécifie l'événement mouseover, qui est déclenché lorsque le pointeur de la souris se déplace sur la partie de l'écran occupée par l'élément hôte.

Contrairement aux liaisons à sens unique, les expressions des liaisons d'événements peuvent modifier l'état de l'application et contenir des opérateurs d'affectation, tels que =. L'expression de la liaison affecte la valeur de la propriété item.name à une variable appelée selectedProduct. La variable selectedProduct est utilisée dans une liaison d'interpolation de chaîne en haut du modèle, comme ceci :

...

<div class="bg-info text-white p-2">

Produit sélectionné : {{selectedProduct ? ? '(None)'}}

</div>

...

La valeur affichée par la liaison d'interpolation de chaîne est mise à jour lorsque la valeur de la variable selectedProduct est modifiée par la liaison d'événement. Il n'est plus nécessaire de lancer manuellement le processus de détection des modifications à l'aide de la méthode ApplicationRef.tick, car les liaisons et les directives présentées dans ce chapitre s'en chargent automatiquement.

**Travailler avec les événements DOM**

Si vous n'êtes pas familier avec les événements qu'un élément HTML peut envoyer, vous trouverez un bon résumé à l'adresse https://developer.mozilla.org/en-US/docs/Web/Events. Il y a beaucoup d'événements, cependant, et tous ne sont pas supportés largement ou de manière cohérente dans tous les navigateurs. Les sections "DOM Events" et "HTML DOM Events" de la page mozilla.org constituent un bon point de départ. Elles définissent les interactions de base qu'un utilisateur a avec un élément (cliquer, déplacer le pointeur, soumettre des formulaires, etc.

Si vous utilisez des événements moins courants, vous devez vous assurer qu'ils sont disponibles et qu'ils fonctionnent comme prévu dans les navigateurs que vous ciblez. L'excellent site http://caniuse.com fournit des détails sur les fonctionnalités mises en œuvre par les différents navigateurs, mais vous devez également procéder à des tests approfondis.

L'expression qui affiche le produit sélectionné utilise l'opérateur de coalescence nul pour s'assurer que l'utilisateur voit toujours un message, même si aucun produit n'est sélectionné. Une approche plus soignée consiste à définir une méthode qui effectue cette vérification, comme le montre l'illustration 12-5.

import { Component } from "@angular/core" ;

import { Model } from "./repository.model" ;

import { Product } from "./product.model" ;

@Component({

selector : "app",

templateUrl : "template.html"

})

export class ProductComponent {

modèle : Model = new Model() ;

getProduct(key : number) : Product | undefined {

return this.model.getProduct(key) ;

}

getProducts() : Produit[] {

return this.model.getProducts() ;

}

selectedProduct : string | undefined ;

getSelected(product : Product) : boolean {

return product.name == this.selectedProduct ;

}

}

Liste 12-5Amélioration du composant dans le fichier component.ts du dossier src/app

J'ai défini une méthode appelée getSelected qui accepte un objet Product et compare son nom à la propriété selectedProduct. Dans le Listing 12-6, la méthode getSelected est utilisée par une liaison de classe pour contrôler l'appartenance à la classe bg-info, qui est une classe Bootstrap qui attribue une couleur d'arrière-plan à un élément.

<div class="p-2">

<div class="bg-info text-white p-2">

Produit sélectionné : {{selectedProduct ? ? '(None)'}}

</div>

<table class="table table-sm table-bordered">

<tr \*ngFor="let item of getProducts() ; let i = index"

[class.bg-info]="getSelected(item)">

<td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>

<td>{{nom de l'article}}</td>

<td>{{article.catégorie}}</td>

<td>{{article.prix}}</td>

</tr>

</table>

</div>

Listing 12-6 Définition de l'appartenance à une classe dans le fichier template.html du dossier src/app

Le résultat est que les éléments tr sont ajoutés à la classe bg-info lorsque la valeur de la propriété selectedProduct correspond à l'objet Product utilisé pour les créer, qui est modifié par la liaison événementielle lorsque l'événement mouseover est déclenché, comme le montre la figure 12-4.

Figure 12-4Mise en évidence des lignes d'un tableau à l'aide d'une liaison événementielle

Cet exemple montre comment l'interaction de l'utilisateur introduit de nouvelles données dans l'application et lance le processus de détection des changements, ce qui amène Angular à réévaluer les expressions utilisées par l'interpolation de chaîne et les liaisons de classe. Ce flux de données est ce qui donne vie aux applications Angular : les liaisons et les directives décrites dans les chapitres 10 et 11 répondent dynamiquement aux changements de l'état de l'application, créant un contenu généré et géré entièrement dans le navigateur.

**Utilisation des données d'événement**

L'exemple précédent a utilisé la liaison événementielle pour relier deux données fournies par le composant : lorsque l'événement mouseevent est déclenché, l'expression de la liaison définit la propriété selectedProduct en utilisant une valeur de données fournie à la directive ngfor par la méthode getProducts du composant.

La liaison d'événement peut également être utilisée pour introduire de nouvelles données dans l'application à partir de l'événement lui-même, en utilisant les détails fournis par le navigateur. La liste 12-7 ajoute un élément input au modèle et utilise la liaison d'événement pour écouter l'événement input, qui est déclenché lorsque le contenu de l'élément input change.

<div class="p-2">

<div class="bg-info text-white p-2">

Produit sélectionné : {{selectedProduct ? ? '(None)'}}

</div>

<table class="table table-sm table-bordered">

<tr \*ngFor="let item of getProducts() ; let i = index"

[class.bg-info]="getSelected(item)">

<td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>

<td>{{nom de l'article}}</td>

<td>{{article.catégorie}}</td>

<td>{{article.prix}}</td>

</tr>

</table>

<div class="form-group">

<label>Nom du produit</label>

<input class="form-control"

(input)="selectedProduct=$any($event).target.value" />

</div>

</div>

Liste 12-7Utilisation d'un objet événement dans le fichier template.html du dossier src/app

Lorsque le navigateur déclenche un événement, il fournit un objet Event qui le décrit. Il existe différents types d'objets Event pour différentes catégories d'événements (événements de souris, événements de clavier, événements de formulaire, etc.), mais tous les événements partagent les trois propriétés décrites dans le tableau 12-3.

Tableau 12-3Les propriétés communes à tous les objets événements DOM

| Name | Description |
| --- | --- |
| type | This property returns a string that identifies the type of event that has been triggered. |
| target | This property returns the object that triggered the event, which will generally be the object that represents the HTML element in the DOM. |
| timeStamp | This property returns a number that contains the time that the event was triggered, expressed as milliseconds since January 1, 1970. |

Nom de l'objet

Description

type

Cette propriété renvoie une chaîne qui identifie le type d'événement qui a été déclenché.

cible

Cette propriété renvoie l'objet qui a déclenché l'événement, qui sera généralement l'objet qui représente l'élément HTML dans le DOM.

horodatage

Cette propriété renvoie un nombre qui contient l'heure à laquelle l'événement a été déclenché, exprimée en millisecondes depuis le 1er janvier 1970.

L'objet Event est affecté à une variable de modèle appelée $event, et l'expression de liaison du Listing 12-7 utilise cette variable pour accéder à l'événement et à sa propriété target, comme ceci :

...

<input class="form-control" (input)="selectedProduct=$any($event).target.value" />

...

Cette expression met en évidence une limitation de la façon dont les types de données sont vérifiés dans les modèles Angular.

Lorsque l'élément de saisie est déclenché, l'API DOM du navigateur crée un objet InputEvent, et c'est cet objet qui est assigné à la variable $event. La propriété InputEvent.target renvoie un objet HTMLInputElement, qui représente dans le DOM l'élément de saisie qui a déclenché l'événement. La propriété HTMLInputElement.value renvoie le contenu de l'élément de saisie. L'association de ces types signifie que la lecture de la valeur de $event.target.value produira le contenu de l'élément d'entrée qui a déclenché l'événement.

Malheureusement, Angular suppose que la variable $event est toujours assignée à un objet Event, qui définit les caractéristiques communes à tous les événements. La propriété Event.target renvoie un objet InputTarget, qui définit uniquement les méthodes nécessaires à la mise en place des gestionnaires d'événements et ne permet pas d'accéder aux fonctionnalités spécifiques aux éléments.

TypeScript a été conçu pour résoudre ce type de problème en utilisant des assertions de type, comme je l'ai expliqué au chapitre 3. Mais Angular ne permet pas l'utilisation du mot-clé as dans les expressions de template, ce qui signifie que je ne peux pas dire aux outils de construction Angular et TypeScript que la variable $event contient un objet InputEvent.

Les modèles Angular prennent en charge la fonction spéciale $any, qui désactive la vérification du type en traitant une valeur comme le type spécial any :

...

<input class="form-control" (input)="selectedProduct=$any($event).target.value" />

...

En passant $event à la fonction $any, je peux lire la propriété target.value sans provoquer d'erreur de compilation. Il convient d'être prudent lors de l'utilisation de la fonction $any, car elle désactive les vérifications de type du compilateur, ce qui peut entraîner des erreurs si les noms de propriétés ou de méthodes spécifiés n'existent pas au moment de l'exécution.

La liaison événementielle a pour effet d'affecter à la variable selectedProduct le contenu de l'élément de saisie après chaque frappe. Lorsque l'utilisateur tape dans l'élément de saisie, le texte saisi est affiché en haut de la fenêtre du navigateur à l'aide de la liaison d'interpolation de chaîne.

La liaison ngClass appliquée aux éléments tr définit la couleur d'arrière-plan des lignes du tableau lorsque la propriété selectedProduct correspond au nom du produit qu'elles représentent. Maintenant que la valeur de la propriété selectedProduct est déterminée par le contenu de l'élément input, la saisie du nom d'un produit entraîne la mise en surbrillance de la ligne correspondante, comme le montre la figure 12-5.

Figure 12-5Utilisation des données d'événements

L'utilisation de différentes liaisons pour travailler ensemble est au cœur du développement efficace d'Angular et permet de créer des applications qui répondent immédiatement à l'interaction de l'utilisateur et aux changements dans le modèle de données.

**Gestion des événements dans le composant**

Bien que les assertions de type ne puissent pas être effectuées dans les modèles, elles peuvent être utilisées dans la classe de composant, comme le montre l'illustration 12-8, qui fournit un moyen de gérer les événements sans avoir besoin d'utiliser le type any.

import { Component } from "@angular/core" ;

import { Model } from "./repository.model" ;

import { Product } from "./product.model" ;

@Component({

selector : "app",

templateUrl : "template.html"

})

export class ProductComponent {

modèle : Model = new Model() ;

getProduct(key : number) : Product | undefined {

return this.model.getProduct(key) ;

}

getProducts() : Produit[] {

return this.model.getProducts() ;

}

selectedProduct : string | undefined ;

getSelected(product : Product) : boolean {

return product.name == this.selectedProduct ;

}

handleInputEvent(ev : Événement) {

if (ev.target instanceof HTMLInputElement) {

this.selectedProduct = ev.target.value

}

}

}

Listing 12-8Définition d'une méthode dans le fichier component.ts du dossier src/app

La méthode handleInputEvent reçoit un objet Event et utilise l'opérateur instanceof pour déterminer si la propriété target de l'événement renvoie un HTMLInputElement. Si c'est le cas, la propriété value est affectée à la propriété selectedProduct. La liste 12-9 met à jour le modèle afin d'utiliser la nouvelle méthode de gestion des événements.

<div class="p-2">

<div class="bg-info text-white p-2">

Produit sélectionné : {{selectedProduct ? ? '(None)'}}

</div>

<table class="table table-sm table-bordered">

<tr \*ngFor="let item of getProducts() ; let i = index"

[class.bg-info]="getSelected(item)">

<td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>

<td>{{nom de l'article}}</td>

<td>{{article.catégorie}}</td>

<td>{{article.prix}}</td>

</tr>

</table>

<div class="form-group">

<label>Nom du produit</label>

<input class="form-control" (input)="handleInputEvent($event)" />

</div>

</div>

Liste 12-9Gestion d'un événement avec une méthode dans le fichier template.html du dossier src/app

L'effet est le même que dans l'exemple précédent, mais l'événement est traité sans désactiver la vérification du type.

**Utilisation des variables de référence gabarit(template)**

Au chapitre 11, j'ai expliqué comment **les variables de gabarit** sont utilisées pour transmettre des données à l'intérieur d'un gabarit, comme la définition d'une variable pour l'objet courant lors de l'utilisation de la directive ngFor. Les variables de référence de modèle sont une forme de variable de modèle qui peut être utilisée pour faire référence à des éléments dans le modèle, comme le montre l'illustration 12-10.

<div class="p-2">

<div class="bg-info text-white p-2">

Produit sélectionné : {{produit.valeur ? ? '(None)'}}

</div>

<table class="table table-sm table-bordered">

<tr \*ngFor="let item of getProducts() ; let i = index"

[class.bg-info]="product.value == item.name">

<td (mouseover)="product.value = item.name ? ? ''">{{i + 1}}</td>

<td>{{nom de l'article}}</td>

<td>{{article.catégorie}}</td> <td>{article.catégorie}</td>

<td>{{article.prix}}</td>

</tr>

</table>

<div class="form-group">

<label>Nom du produit</label>

<input #product class="form-control" (input)="false" />

</div>

</div>

Liste 12-10Utilisation d'une variable de modèle dans le fichier template.html du dossier src/app

Les variables de référence sont définies à l'aide du caractère #, suivi du nom de la variable. Dans la liste, j'ai défini une variable appelée product comme suit :

...

<input #product class="form-control" (input)="false" />

...

Lorsqu'Angular rencontre une variable de référence dans un modèle, il fixe sa valeur à l'élément auquel elle a été appliquée. Dans cet exemple, la variable de référence product est assignée à l'objet qui représente l'élément input dans le DOM, l'objet HTMLInputElement. Les variables de référence peuvent être utilisées par d'autres liaisons dans le même modèle. Ceci est démontré par la liaison d'interpolation de chaîne, qui utilise également la variable produit, comme ceci :

...

Produit sélectionné : {{produit.valeur ? ? '(None)'}}

...

Cette liaison affiche la propriété de valeur définie par l'élément HTMLInputElement qui a été affecté à la variable product ou la chaîne (None) si la propriété de valeur renvoie un résultat nul ou indéfini. Les variables de modèle peuvent également être utilisées pour modifier l'état de l'élément, comme le montre cette liaison :

...

<td (mouseover)="product.value = item.name ? ? ''">{{i + 1}}</td>

...

La liaison événementielle répond à l'événement mouseover en définissant la propriété value sur l'élément HTMLInputElement qui a été assigné à la variable product. Le résultat est que le passage de la souris sur l'un des éléments td de la première colonne du tableau met à jour le contenu de l'élément d'entrée.

Cet exemple comporte un aspect gênant, à savoir la liaison de l'événement "input" à l'élément "input".

...

<input #product class="form-control" (input)="false" />

...

Angular ne mettra pas à jour les liaisons de données dans le modèle lorsque l'utilisateur modifie le contenu de l'élément input à moins qu'il n'y ait une liaison d'événement sur cet élément. Définir la liaison à false donne à Angular quelque chose à évaluer juste pour que le processus de mise à jour commence et distribue le contenu actuel de l'élément d'entrée dans tout le modèle. Il s'agit d'une bizarrerie qui étire un peu trop le rôle d'une variable de référence de modèle et ce n'est pas quelque chose que vous aurez besoin de faire dans la plupart des projets réels. La plupart des liaisons de données reposent sur des variables définies par le composant du modèle, comme nous l'avons vu dans la section précédente.

**Filtrage des événements clés**

L'événement input est déclenché chaque fois que le contenu de l'élément input est modifié. Cela permet d'obtenir un ensemble de changements immédiats et réactifs, mais ce n'est pas ce dont toutes les applications ont besoin, en particulier si la mise à jour de l'état de l'application implique des opérations coûteuses.

La liaison événementielle dispose d'un support intégré permettant d'être plus sélectif lors de la liaison avec les événements clavier, ce qui signifie que les mises à jour ne seront effectuées que lorsqu'une touche spécifique est enfoncée. Voici une liaison qui répond à chaque frappe de touche :

...

<input #product class="form-control" (keyup)="selectedProduct=product.value" />

...

L'événement keyup est un événement DOM standard, et le résultat est que l'application est mise à jour lorsque l'utilisateur relâche chaque touche pendant qu'il tape dans l'élément de saisie. Je peux être plus précis quant à la touche qui m'intéresse en spécifiant son nom dans le cadre de la liaison d'événement, comme ceci :

...

<input #product class="form-control"

(keyup.enter)="selectedProduct=product.value" />

...

La touche à laquelle la liaison répondra est spécifiée en ajoutant un point après le nom de l'événement DOM, suivi du nom de la touche. Cette liaison est pour la touche Entrée, et le résultat est que les changements dans l'élément d'entrée ne seront pas poussés dans le reste de l'application jusqu'à ce que cette touche soit pressée.

**Utilisation de liaisons de données bidirectionnelles**

Les liaisons peuvent être combinées pour créer un flux de données bidirectionnel pour un seul élément, ce qui permet au document HTML de réagir lorsque le modèle d'application change et à l'application de réagir lorsque l'élément émet un événement, comme le montre la liste 12-11.

<div class="p-2">

<div class="bg-info text-white p-2">

Produit sélectionné : {{ selectedProduct ? ? '(None)' }}

</div>

<table class="table table-sm table-bordered">

<tr \*ngFor="let item of getProducts() ; let i = index"

[class.bg-info]="getSelected(item)">

<td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>

<td>{{nom de l'article}}</td>

<td>{{article.catégorie}}</td>

<td>{{article.prix}}</td>

</tr>

</table>

<div class="form-group">

<label>Nom du produit</label>

<input class="form-control"

(input)="selectedProduct=$any($event).target.value"

[valeur]="produit sélectionné ? ? ''" />

</div>

<div class="form-group">

<label>Nom du produit</label>

<input class="form-control"

(input)="selectedProduct=$any($event).target.value"

[valeur]="produit sélectionné ? ? ''" />

</div>

</div>

Listing 12-11Création d'une liaison bidirectionnelle dans le fichier template.html du dossier src/app

Chaque élément d'entrée possède **une liaison d'événement** et une **liaison de propriété**. La liaison d'événement répond à l'événement de saisie en mettant à jour la propriété selectedProduct du composant. La liaison de propriété lie la valeur de la propriété selectedProduct à la propriété value de l'élément.

Le résultat est que le contenu des deux éléments de saisie est synchronisé et que la modification de l'un entraîne la mise à jour de l'autre. De plus, étant donné que d'autres liaisons du modèle dépendent de **la propriété selectedProduct**, la modification du contenu d'un élément de saisie modifie également les données affichées par la liaison d'interpolation de chaîne et la ligne de tableau en surbrillance, comme le montre la figure 12-6.

Figure 12-6Création d'une liaison de données bidirectionnelle

Cet exemple prend tout son sens lorsqu'on l'expérimente dans le navigateur. Saisissez du texte dans l'un des éléments de saisie et vous verrez le même texte s'afficher dans l'autre élément de saisie et dans l'élément div dont le contenu est géré par la liaison par interpolation de chaînes. Si vous saisissez le nom d'un produit dans l'un des éléments de saisie, tel que Kayak ou Gilet de sauvetage, vous verrez également la ligne correspondante dans le tableau en surbrillance.

La liaison événementielle pour l'événement "mouseover" reste active, ce qui signifie que lorsque vous déplacez le pointeur de la souris sur la première ligne du tableau, les modifications apportées à la valeur selectedProduct entraînent l'affichage du nom du produit dans les éléments de saisie.

**Utilisation de la directive ngModel**

La directive ngModel est utilisée pour simplifier les liaisons bidirectionnelles afin de ne pas avoir à appliquer à la fois **une liaison d'événement** et **une liaison de propriété** au même élément. La liste 12-12 montre comment remplacer les liaisons séparées par la directive ngModel.

<div class="p-2">

<div class="bg-info text-white p-2">

Produit sélectionné : {{ selectedProduct ? ? '(None)' }}

</div>

<table class="table table-sm table-bordered">

<tr \*ngFor="let item of getProducts() ; let i = index"

[class.bg-info]="getSelected(item)">

<td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>

<td>{{nom de l'article}}</td>

<td>{{article.catégorie}}</td>

<td>{{article.prix}}</td>

</tr>

</table>

<div class="form-group">

<label>Nom du produit</label>

<input class="form-control" [(ngModel)]="produit sélectionné" />

</div>

<div class="form-group">

<label>Nom du produit</label>

<input class="form-control" [(ngModel)]="produit sélectionné" />

</div>

</div>

Liste 12-12Utilisation de la directive ngModel dans le fichier template.html du dossier src/app

L'utilisation de la directive **ngModel** nécessite de combiner la syntaxe des liaisons de propriétés et d'événements, comme l'illustre la figure 12-7.

Figure 12-7Anatomie d'une liaison bidirectionnelle de données

Une combinaison de crochets carrés et ronds est utilisée pour indiquer une liaison de données bidirectionnelle, les crochets ronds étant placés à l'intérieur des crochets carrés : [( et )]. L'équipe de développement d'Angular appelle cela la liaison "banane dans une boîte" car c'est à cela que ressemblent les crochets et les parenthèses lorsqu'ils sont placés comme ceci [()]. Enfin, en quelque sorte.

**La cible de la liaison** est la directive **ngModel**, qui est incluse dans Angular pour simplifier la création de liaisons de données bidirectionnelles sur les éléments de formulaire, tels que les éléments d'entrée utilisés dans l'exemple.

L'expression d'une liaison de données bidirectionnelle est le nom d'une propriété, qui est utilisée pour configurer les différentes liaisons en coulisses. Lorsque le contenu de l'élément d'entrée change, le nouveau contenu est utilisé pour mettre à jour la valeur de la propriété selectedProduct. De même, lorsque la valeur de la propriété selectedProduct change, elle est utilisée pour mettre à jour le contenu de l'élément.

La directive ngModel connaît la combinaison d'événements et de propriétés que les éléments HTML standard définissent. **En coulisses, une liaison d'événement est appliquée à l'événement d'entrée et une liaison de propriété est appliquée à la propriété de valeur.**

ConseilVous devez vous rappeler d'utiliser à la fois des crochets et des parenthèses avec la liaison ngModel. Si vous n'utilisez que des parenthèses (ngModel), vous définissez une liaison d'événement pour un événement appelé ngModel, qui n'existe pas. Le résultat est un élément qui ne sera pas mis à jour ou qui ne mettra pas à jour le reste de l'application. Vous pouvez utiliser la directive ngModel avec seulement des crochets - [ngModel]- et Angular définira la valeur initiale de l'élément mais n'écoutera pas les événements, ce qui signifie que les modifications apportées par l'utilisateur ne seront pas automatiquement reflétées dans le modèle de l'application.

**Travailler avec des formulaires**

La plupart des applications web s'appuient sur des formulaires pour recevoir des données de la part des utilisateurs, et la liaison bidirectionnelle ngModel décrite dans la section précédente constitue la base de l'utilisation des formulaires dans les applications Angular. Dans cette section, je crée un formulaire qui permet de créer de nouveaux produits et de les ajouter au modèle de données de l'application, puis je décris certaines des fonctionnalités de formulaire plus avancées fournies par Angular.

**Ajout d'un formulaire à l'application d'exemple**

La liste 12-13 montre certaines améliorations apportées au composant qui sera utilisé lors de la création du formulaire et supprime certaines fonctionnalités qui ne sont plus nécessaires.

import { Component } from "@angular/core" ;

import { Model } from "./repository.model" ;

import { Product } from "./product.model" ;

@Component({

selector : "app",

templateUrl : "template.html"

})

export class ProductComponent {

modèle : Model = new Model() ;

getProduct(key : number) : Product | undefined {

return this.model.getProduct(key) ;

}

getProducts() : Produit[] {

return this.model.getProducts() ;

}

// selectedProduct : string | undefined ;

// getSelected(product : Product) : boolean {

// return product.name == this.selectedProduct ;

// }

// handleInputEvent(ev : Event) {

// if (ev.target instanceof HTMLInputElement) {

// this.selectedProduct = ev.target.value

// }

// }

nouveauProduit : Produit = nouveau Produit() ;

get jsonProduct() {

return JSON.stringify(this.newProduct) ;

}

addProduct(p : Produit) {

console.log("Nouveau produit : " + this.jsonProduct) ;

}

}

Listing 12-13Améliorer le composant dans le fichier component.ts du dossier src/app

La liste ajoute une nouvelle propriété appelée newProduct, qui sera utilisée pour stocker les données saisies dans le formulaire par l'utilisateur. Il y a également une propriété jsonProduct avec un getter qui renvoie une représentation JSON de la propriété newProduct et qui sera utilisée dans le modèle pour montrer l'effet des liaisons bidirectionnelles. (Je ne peux pas créer une représentation JSON d'un objet directement dans le modèle parce que l'objet JSON est défini dans l'espace de noms global qui, comme je l'ai expliqué au chapitre 11, n'est pas accessible directement à partir d'expressions de modèle).

Le dernier ajout est une méthode addProduct qui écrit la valeur de la méthode jsonProduct dans la console ; cela me permettra de démontrer quelques fonctionnalités de base liées au formulaire avant d'ajouter la prise en charge de la mise à jour du modèle de données plus tard dans le chapitre.

Dans la liste 12-14, le contenu du modèle a été remplacé par une série d'éléments de saisie pour chacune des propriétés définies par la classe Product.

<div class="p-2">

<div class="bg-info text-white mb-2 p-2">Données du modèle : {{jsonProduct}}</div>

<div class="form-group">

<label>Nom</label>

<input class="form-control" [(ngModel)]="nouveauProduit.nom" />

</div>

<div class="form-group">

<label>Catégorie</label>

<input class="form-control" [(ngModel)]="nouveauProduit.catégorie" />

</div>

<div class="form-group">

<label>Prix</label>

<input class="form-control" [(ngModel)]="nouveauProduit.prix" />

</div>

<button class="btn btn-primary mt-2" (click)="addProduct(newProduct)">

Créer

</button>

</div>

Liste 12-14Remplacer le contenu du fichier template.html dans le dossier src/app

Chaque élément de saisie est regroupé avec une étiquette et contenu dans un élément div, qui est stylisé à l'aide de la classe Bootstrap form-group. Les éléments de saisie individuels sont affectés à la classe Bootstrap form-control pour gérer la mise en page et le style.

La liaison ngModel a été appliquée à chaque élément d'entrée pour créer une liaison bidirectionnelle avec la propriété correspondante de l'objet newProduct du composant, comme ceci :

...

<input class="form-control" [(ngModel)]="newProduct.name" />

...

Il y a également un élément bouton, qui a une liaison pour l'événement click qui appelle la méthode addProduct du composant, en transmettant la valeur newProduct en tant qu'argument.

...

<button class="btn btn-primary" (click)="addProduct(newProduct)">Créer</button>

...

Enfin, une liaison d'interpolation de chaîne est utilisée pour afficher une représentation JSON de la propriété newProduct du composant en haut du modèle, comme ceci :

...

<div class="bg-info text-white mb-2 p-2">Données du modèle : {{jsonProduct}}</div>

...

Le résultat global, illustré à la figure 12-8, est un ensemble d'éléments d'entrée qui mettent à jour les propriétés d'un objet Product géré par le composant, ce qui se reflète immédiatement dans les données JSON.

Figure 12-8Utilisation des éléments du formulaire pour créer un nouvel objet dans le modèle de données

Lorsque l'on clique sur le bouton Créer, la représentation JSON de la propriété newProduct du composant est écrite dans la console JavaScript du navigateur, ce qui produit un résultat comme celui-ci :

Nouveau produit : {"name":"Running Shoes","category":"Running","price":"120.23"}

**Ajout de la validation des données du formulaire**

Pour l'instant, n'importe quelle donnée peut être saisie dans les éléments d'entrée du formulaire. La validation des données est essentielle dans les applications web, car les utilisateurs entrent une gamme surprenante de valeurs de données, soit par erreur, soit parce qu'ils veulent arriver à la fin du processus le plus rapidement possible et entrer des valeurs de déchets pour continuer.

Angular fournit un système extensible de validation du contenu des éléments de formulaire, basé sur l'approche utilisée par le standard HTML5. Le tableau 12-4 liste les attributs que vous pouvez ajouter aux éléments de saisie, chacun d'entre eux définissant une règle de validation.

Tableau 12-4Les attributs de validation intégrés à Angular

Attribut

Description

courriel

Cet attribut est utilisé pour spécifier une adresse électronique bien formatée.

requis

Cet attribut est utilisé pour spécifier une valeur qui doit être fournie.

longueur minimale

Cet attribut est utilisé pour spécifier un nombre minimum de caractères.

longueur maximale

Cet attribut est utilisé pour spécifier un nombre maximum de caractères. Ce type de validation ne peut pas être appliqué directement aux éléments de formulaire car il entre en conflit avec l'attribut HTML5 du même nom. Il peut être utilisé avec les formulaires basés sur un modèle, qui sont décrits plus loin dans ce chapitre.

min

Cet attribut est utilisé pour spécifier une valeur minimale.

max

Cet attribut est utilisé pour spécifier une valeur maximale.

modèle

Cet attribut est utilisé pour spécifier une expression régulière à laquelle la valeur fournie par l'utilisateur doit correspondre.

Ces attributs vous sont peut-être familiers car ils font partie de la spécification HTML, mais Angular s'appuie sur ces propriétés avec quelques fonctionnalités supplémentaires. Le Listing 12-15 supprime tous les éléments d'entrée sauf un pour démontrer le processus d'ajout de la validation au formulaire aussi simplement que possible. (Je rétablis les éléments manquants à la fin du chapitre).

<div class="p-2">

<div class="bg-info text-white mb-2 p-2">Données du modèle : {{jsonProduct}}</div>

<form (ngSubmit)="addProduct(newProduct)">

<div class="form-group">

<label>Nom</label>

<input class="form-control"

name="nom"

[(ngModel)]="nouveauProduit.nom"

obligatoire

minlength="5"

pattern="^[A-Za-z ]+$" />

</div>

<button class="btn btn-primary mt-2" type="submit">

Créer

</button>

</form>

</div>

Liste 12-15Ajouter la validation du formulaire dans le fichier template.html du dossier src/app

Angular exige que les éléments validés définissent l'attribut name, qui est utilisé pour identifier l'élément dans le système de validation. Puisque cet élément de saisie est utilisé pour capturer la valeur de la propriété Product.name, l'attribut name de l'élément a été défini à name.

Cette liste ajoute trois des attributs de validation à l'élément input. L'attribut required spécifie que l'utilisateur doit fournir une valeur, l'attribut minlength spécifie qu'il doit y avoir au moins trois caractères et l'attribut pattern spécifie que seuls les caractères alphabétiques et les espaces sont autorisés.

Enfin, remarquez qu'un élément de formulaire a été ajouté au modèle. Bien que vous puissiez utiliser des éléments de saisie indépendamment, les fonctions de validation d'Angular ne fonctionnent que lorsqu'un élément de formulaire est présent, et Angular signalera une erreur si vous ajoutez la directive ngControl à un élément qui n'est pas contenu dans un formulaire.

Lors de l'utilisation d'un élément de formulaire, la convention est d'utiliser une liaison d'événement pour un événement spécial appelé ngSubmit comme ceci :

...

<form (ngSubmit)="addProduct(newProduct)">

...

La liaison ngSubmit gère l'événement de soumission de l'élément de formulaire. Si vous préférez, vous pouvez obtenir le même effet en vous liant à l'événement click de chaque bouton du formulaire.

**Stylisation des éléments à l'aide des classes de validation**

Une fois que vous avez enregistré les modifications apportées au modèle dans la liste 12-15 et que le navigateur a rechargé le code HTML, cliquez avec le bouton droit de la souris sur l'élément de saisie dans la fenêtre du navigateur et sélectionnez Inspecter ou Inspecter l'élément dans la fenêtre contextuelle. Le navigateur affiche la représentation HTML de l'élément dans la fenêtre Developer Tools, et vous voyez que l'élément input a été ajouté à trois classes, comme ceci :

...

<input name="name" required="" minlength="5" pattern="^[A-Za-z ]+$"

class="form-control ng-pristine ng-invalid ng-touched" ng-reflect-required=""

ng-reflect-minlength="5" ng-reflect-pattern="^[A-Za-z ]+$" ng-reflect-name="name">

...

Les classes auxquelles un élément d'entrée est affecté fournissent des détails sur son état de validation. Il existe trois paires de classes de validation, décrites dans le tableau 12-5. Les éléments seront toujours membres d'une classe de chaque paire, pour un total de trois classes. Les mêmes classes sont appliquées à l'élément de formulaire pour montrer l'état de validation global de tous les éléments qu'il contient. Lorsque l'état de l'élément d'entrée change, la directive ngControl modifie automatiquement les classes pour les éléments individuels et l'élément de formulaire.

Tableau 12-5Les classes de validation du formulaire Angular

Nom

Description

ng-untouched ng-touched

Un élément est affecté à la classe ng-untouched s'il n'a pas été visité par l'utilisateur, ce qui se fait généralement en parcourant les champs du formulaire à l'aide de la tabulation. Une fois que l'utilisateur a visité un élément, celui-ci est ajouté à la classe ng-touched.

ng-pristine ng-dirty

Un élément est affecté à la classe ng-pristine si son contenu n'a pas été modifié par l'utilisateur et à la classe ng-dirty dans le cas contraire. Une fois que le contenu a été modifié, un élément reste dans la classe ng-dirty, même si l'utilisateur revient ensuite au contenu précédent.

ng-valid ng-invalid

Un élément est affecté à la classe ng-valid si son contenu répond aux critères définis par les règles de validation qui lui ont été appliquées et à la classe ng-invalid dans le cas contraire.

ng-pending

Les éléments sont affectés à la classe ng-pending lorsque leur contenu est validé de manière asynchrone. Voir les chapitres 21 et 22 pour plus de détails.

Ces classes peuvent être utilisées pour styliser les éléments du formulaire afin de fournir à l'utilisateur un retour d'information sur la validation. Ajoutez les styles présentés dans la liste 12-16 au fichier styles.css dans le dossier src.

html, body { height : 100% ; }

body { margin : 0 ; font-family : Roboto, "Helvetica Neue", sans-serif ; }

input.ng-dirty.ng-invalid { border : 2px solid #ff0000 }

input.ng-dirty.ng-valid { border : 2px solid #6bc502 }

Liste 12-16Définition des styles de rétroaction de validation dans le fichier styles.css du dossier src/app

Ces styles définissent des bordures vertes et rouges pour les éléments de saisie dont le contenu a été modifié et est valide (et qui appartiennent donc aux classes ng-dirty et ng-valid) et dont le contenu est invalide (et qui appartiennent donc aux classes ng-dirty et ng-invalid). L'utilisation de la classe ng-dirty signifie que l'apparence des éléments ne sera pas modifiée tant que l'utilisateur n'aura pas saisi de contenu.

Angular valide le contenu et modifie les appartenances de classe des éléments de saisie après chaque frappe ou changement de focus. Le navigateur détecte les modifications apportées aux éléments et applique les styles de manière dynamique, ce qui permet aux utilisateurs de bénéficier d'un retour d'information sur la validation lorsqu'ils saisissent des données dans le formulaire, comme le montre la figure 12-9.

Figure 12-9Fournir un retour d'information sur la validation

Lorsque je commence à taper, l'élément de saisie apparaît comme invalide car il n'y a pas assez de caractères pour satisfaire l'attribut minlength. Lorsqu'il y a cinq caractères, la bordure devient verte, ce qui indique que les données sont valides. Lorsque je tape le caractère 2, la bordure redevient rouge car l'attribut pattern est défini de manière à n'autoriser que les lettres et les espaces.

ConseilSi vous regardez les données JSON en haut de la page dans la figure 12-9, vous verrez que les liaisons de données sont toujours mises à jour, même lorsque les valeurs des données ne sont pas valides. La validation fonctionne parallèlement aux liaisons de données, et vous ne devez pas agir sur les données du formulaire sans vérifier que l'ensemble du formulaire est valide, comme décrit dans la section "Validation de l'ensemble du formulaire".

**Affichage des messages de validation au niveau du champ**

L'utilisation de couleurs pour fournir des messages de validation indique à l'utilisateur que quelque chose ne va pas, mais ne lui donne aucune indication sur ce qu'il doit faire. La directive ngModel permet d'accéder à l'état de validation des éléments auxquels elle s'applique, ce qui peut être utilisé pour afficher des conseils à l'utilisateur. La liste 12-17 ajoute des messages de validation pour chacun des attributs appliqués à l'élément input en utilisant le support fourni par la directive ngModel.

<div class="p-2">

<div class="bg-info text-white mb-2 p-2">Données du modèle : {{jsonProduct}}</div>

<form (ngSubmit)="addProduct(newProduct)">

<div class="form-group">

<label>Nom</label>

<input class="form-control"

name="nom"

[(ngModel)]="nouveauProduit.nom"

#nom="ngModel"

obligatoire

minlength="5"

pattern="^[A-Za-z ]+$" />

<ul class="text-danger list-unstyled mt-1"

\*ngIf="name.dirty && name.invalid">

<li \*ngIf="name.errors ?.['required']">

Vous devez saisir un nom de produit

</li>

<li \*ngIf="name.errors ?.['pattern']">

Les noms de produits ne peuvent contenir que des lettres et des espaces

</li>

<li \*ngIf="name.errors ?.['minlength']">

Les noms de produits doivent être au moins

{{ name.errors ?.['minlength'].requiredLength }} caractères

</li>

</ul>

</div>

<button class="btn btn-primary mt-2" type="submit">

Créer

</button>

</form>

</div>

Liste 12-17Ajouter des messages de validation dans le fichier template.html du dossier src/app

Pour que la validation fonctionne, je dois créer une variable de référence de modèle pour accéder à l'état de validation dans les expressions, ce que je fais comme suit :

...

<input class="form-control" name="name" [(ngModel)]="newProduct.name"

#name="ngModel" required minlength="5" pattern="^[A-Za-z ]+$"/>

...

Je crée une variable de référence de modèle appelée name et je lui attribue la valeur ngModel. Cette utilisation d'une valeur ngModel est un peu déroutante : il s'agit d'une fonctionnalité fournie par la directive ngModel pour donner accès au statut de validation. Cela aura plus de sens lorsque vous aurez lu les chapitres 13 et 16, dans lesquels j'explique comment créer des directives personnalisées et vous verrez comment elles permettent d'accéder à leurs fonctionnalités. Pour ce chapitre, il suffit de savoir que pour afficher les messages de validation, il faut créer une variable de référence de modèle et lui assigner ngModel pour accéder aux données de validation de l'élément d'entrée. L'objet assigné à la variable de référence du modèle définit les propriétés décrites dans le tableau 12-6. Toutes les propriétés décrites dans le tableau sont nullables.

Tableau 12-6 Propriétés de l'objet Validation

Nom de l'objet

Description

chemin

Cette propriété renvoie le nom de l'élément.

valide

Cette propriété renvoie la valeur true si le contenu de l'élément est valide et false dans le cas contraire.

invalide

Cette propriété renvoie la valeur true si le contenu de l'élément est invalide et false dans le cas contraire.

immaculé

Cette propriété renvoie la valeur "true" si le contenu de l'élément n'a pas été modifié.

sale

Cette propriété renvoie un résultat positif si le contenu de l'élément a été modifié.

touché

Cette propriété renvoie la valeur "true" si l'utilisateur a visité l'élément.

intact

Cette propriété renvoie la valeur "true" si l'utilisateur n'a pas visité l'élément.

erreurs

Cette propriété renvoie un objet ValidationErrors dont les propriétés correspondent à chaque attribut pour lequel il y a une erreur de validation.

valeur

Cette propriété renvoie la valeur de l'élément, qui est utilisée lors de la définition de règles de validation personnalisées, comme décrit dans la section "Création de validateurs de formulaires personnalisés".

La liste 12-17 affiche les messages de validation dans une liste. La liste ne doit être affichée que s'il y a au moins une erreur de validation, j'ai donc appliqué la directive ngIf à l'élément ul, avec une expression qui utilise les propriétés dirty et invalid, comme ceci :

...

<ul class="text-danger list-unstyled mt-1" \*ngIf="name.dirty && name.invalid">

...

À l'intérieur de l'élément ul, il y a un élément li qui correspond à chaque erreur de validation qui peut se produire. Chaque élément li possède une directive ngIf qui utilise la propriété errors décrite dans le tableau 12-6, comme ceci :

...

<li \*ngIf="name.errors ?.['required']">

Vous devez saisir un nom de produit

</li>

...

La propriété errors.[required ] ne sera définie que si le contenu de l'élément a échoué au contrôle de validation requis, ce qui lie la visibilité de l'élément li au résultat de ce contrôle de validation.

Chaque propriété définie par l'objet errors renvoie un objet dont les propriétés fournissent des détails sur la raison pour laquelle le contenu a échoué au contrôle de validation de son attribut, ce qui peut être utilisé pour rendre les messages de validation plus utiles à l'utilisateur. Le tableau 12-7 décrit les propriétés d'erreur fournies pour chaque attribut.

Tableau 12-7Propriétés de description des erreurs de validation du formulaire Angular

Nom de l'erreur

Description

courriel

Cette propriété renvoie un message vrai si l'attribut email a été appliqué à l'élément d'entrée. Ce n'est pas particulièrement utile car cela peut être déduit du fait que la propriété existe.

requis

Cette propriété renvoie un message vrai si l'attribut requis a été appliqué à l'élément d'entrée. Ce n'est pas particulièrement utile car cela peut être déduit du fait que la propriété existe.

minlength.requiredLength

Cette propriété renvoie le nombre de caractères requis pour satisfaire l'attribut minlength.

minlength.actualLength

Cette propriété renvoie le nombre de caractères saisis par l'utilisateur.

maxlength.requiredLength

Cette propriété renvoie le nombre de caractères requis pour satisfaire l'attribut maxlength.

maxlength.actualLength

Cette propriété renvoie le nombre de caractères saisis par l'utilisateur.

min.réel

Cette propriété renvoie la valeur saisie par l'utilisateur.

min.min

Cette propriété renvoie la valeur minimale requise pour satisfaire l'attribut min.

max.réel

Cette propriété renvoie la valeur saisie par l'utilisateur.

max.max

Cette propriété renvoie la valeur minimale requise pour satisfaire l'attribut max.

pattern.requiredPattern

Cette propriété renvoie l'expression régulière qui a été spécifiée à l'aide de l'attribut pattern.

modèle.valeuractuelle

Cette propriété renvoie le contenu de l'élément.

Ces propriétés ne sont pas affichées directement à l'utilisateur, qui ne comprendra probablement pas un message d'erreur contenant une expression régulière, mais elles peuvent être utiles au cours du développement pour résoudre les problèmes de validation. L'exception est la propriété minlength.requiredLength, qui peut être utile pour éviter la duplication de la valeur assignée à l'attribut minlength sur l'élément, comme ceci :

...

<li \*ngIf="name.errors ?.['minlength']">

Les noms de produits doivent être au moins

{{ name.errors ?.['minlength'].requiredLength }} caractères

</li>

...

Le résultat global est un ensemble de messages de validation qui s'affichent dès que l'utilisateur commence à modifier l'élément de saisie et qui changent pour refléter chaque nouvelle frappe, comme l'illustre la figure 12-10.

Figure 12-10Affichage des messages de validation

Utilisation du composant pour afficher les messages de validation

L'inclusion d'éléments distincts pour toutes les erreurs de validation possibles devient rapidement verbeuse dans les formulaires complexes. Une meilleure approche consiste à ajouter de la logique au composant pour préparer les messages de validation dans une méthode, qui peut ensuite être affichée à l'utilisateur par le biais de la directive ngFor du modèle. L'illustration 12-18 montre l'ajout d'une méthode de composant qui accepte l'état de validation d'un élément d'entrée et produit un tableau de messages de validation.

import { Component } from "@angular/core" ;

import { Model } from "./repository.model" ;

import { Product } from "./product.model" ;

import { NgModel, ValidationErrors } from "@angular/forms" ;

@Component({

selector : "app",

templateUrl : "template.html"

})

export class ProductComponent {

modèle : Model = new Model() ;

getProduct(key : number) : Product | undefined {

return this.model.getProduct(key) ;

}

getProducts() : Produit[] {

return this.model.getProducts() ;

}

newProduct : Produit = nouveau Produit() ;

get jsonProduct() {

return JSON.stringify(this.newProduct) ;

}

addProduct(p : Produit) {

console.log("Nouveau produit : " + this.jsonProduct) ;

}

getMessages(errs : ValidationErrors | null, name : string) : string[] {

let messages : string[] = [] ;

for (let errorName in errs) {

switch (errorName) {

case "required" :

messages.push(`Vous devez entrer un ${nom}`) ;

break ;

case "minlength" :

messages.push(`Un ${nom} doit être au moins égal à

${errs['minlength'].requiredLength}

caractères`) ;

break ;

case "pattern" :

messages.push(`Le ${nom} contient des

caractères illégaux`) ;

break ;

}

}

return messages ;

}

getValidationMessages(state : NgModel, thingName? : string) {

let thing : string = state.path ?.[0] ? ? thingName ;

return this.getMessages(state.errors, thing)

}

}

Listing 12-18Génération des messages de validation dans le fichier component.ts du dossier src/app

Les méthodes getValidationMessages et getMessages utilisent les propriétés décrites dans le tableau 12-6 pour produire des messages de validation pour chaque erreur, en les renvoyant dans un tableau de chaînes. Pour rendre ce code aussi largement applicable que possible, la méthode accepte une valeur décrivant l'élément de données qu'un élément d'entrée est censé collecter auprès de l'utilisateur, qui est ensuite utilisée pour générer des messages d'erreur, comme ceci :

...

messages.push(`Vous devez entrer un ${nom}`) ;

...

Il s'agit d'un exemple de la fonction d'interpolation de chaînes de JavaScript, qui permet de définir des chaînes comme des modèles, sans avoir à utiliser l'opérateur + pour inclure des valeurs de données. Notez que la chaîne de caractères du modèle est indiquée par des caractères backtick (le caractère ` et non le caractère JavaScript '). La méthode getValidationMessages utilise par défaut la propriété path comme chaîne descriptive si aucun argument n'est reçu lorsque la méthode est invoquée, comme ceci :

...

let thing : string = state.path ?.[0] ? ? thingName ;

...

La liste 12-19 montre comment getValidationMessages peut être utilisé dans le modèle pour générer des messages d'erreur de validation pour l'utilisateur sans avoir à définir des éléments et des liaisons distincts pour chacun d'entre eux.

<div class="p-2">

<div class="bg-info text-white mb-2 p-2">Données du modèle : {{jsonProduct}}</div>

<form (ngSubmit)="addProduct(newProduct)">

<div class="form-group">

<label>Nom</label>

<input class="form-control"

name="nom"

[(ngModel)]="nouveauProduit.nom"

#nom="ngModel"

obligatoire

minlength="5"

pattern="^[A-Za-z ]+$" />

<ul class="text-danger list-unstyled mt-1"

\*ngIf="name.dirty && name.invalid">

<li \*ngFor="let error of getValidationMessages(name)">

{{error}}

</li>

</ul>

</div>

<button class="btn btn-primary mt-2" type="submit">

Créer

</button>

</form>

</div>

Figure 12-19 - Obtention de messages de validation dans le fichier template.html du dossier src/app

Il n'y a pas de changement visuel, mais la même méthode peut être utilisée pour produire des messages de validation pour plusieurs éléments, ce qui permet d'obtenir un modèle plus simple, plus facile à lire et à maintenir.

Validation de l'ensemble du formulaire

L'affichage de messages d'erreur de validation pour des champs individuels est utile car il permet de mettre en évidence les problèmes à résoudre. Mais il peut également être utile de valider l'ensemble du formulaire. Il faut veiller à ne pas submerger l'utilisateur de messages d'erreur jusqu'à ce qu'il essaie de soumettre le formulaire, auquel cas un résumé des problèmes peut être utile. En guise de préparation, le Listing 12-20 ajoute deux nouveaux membres au composant.

import { Component } from "@angular/core" ;

import { Model } from "./repository.model" ;

import { Product } from "./product.model" ;

import { NgModel, ValidationErrors, NgForm } from "@angular/forms" ;

@Component({

selector : "app",

templateUrl : "template.html"

})

export class ProductComponent {

modèle : Modèle = nouveau Modèle() ;

// ...autres méthodes omises par souci de concision...

formSubmitted : boolean = false ;

submitForm(form : NgForm) {

this.formSubmitted = true ;

if (form.valid) {

this.addProduct(this.newProduct) ;

this.newProduct = new Product() ;

form.resetForm() ;

this.formSubmitted = false ;

}

}

}

Listing 12-20Amélioration du composant dans le fichier component.ts du dossier src/app

La propriété formSubmitted sera utilisée pour indiquer si le formulaire a été soumis et pour empêcher la validation de l'ensemble du formulaire tant que l'utilisateur n'a pas essayé de le soumettre.

La méthode submitForm est invoquée lorsque l'utilisateur soumet le formulaire et reçoit un objet NgForm comme argument. Cet objet représente le formulaire et définit l'ensemble des propriétés de validation ; ces propriétés sont utilisées pour décrire l'état de validation global du formulaire de sorte que, par exemple, la propriété invalid sera vraie s'il y a des erreurs de validation sur l'un des éléments contenus dans le formulaire. Outre la propriété de validation, NgForm fournit la méthode resetForm, qui réinitialise l'état de validation du formulaire et le ramène à son état d'origine.

En effet, l'ensemble du formulaire est validé lorsque l'utilisateur effectue une validation, et s'il n'y a pas d'erreurs de validation, un nouvel objet est ajouté au modèle de données avant que le formulaire ne soit réinitialisé afin qu'il puisse être utilisé à nouveau. La liste 12-21 montre les modifications à apporter au modèle pour tirer parti de ces nouvelles fonctionnalités et mettre en œuvre la validation à l'échelle du formulaire.

<div class="p-2">

<form #form="ngForm" (ngSubmit)="submitForm(form)">

<div class="bg-danger text-white p-2 mb-2"

\*ngIf="formSubmitted && form.invalid">

Il y a des problèmes avec le formulaire

</div>

<div class="form-group">

<label>Nom</label>

<input class="form-control"

name="nom"

[(ngModel)]="nouveauProduit.nom"

#nom="ngModel"

obligatoire

minlength="5"

pattern="^[A-Za-z ]+$" />

<ul class="text-danger list-unstyled mt-1"

\*ngIf="(formSubmitted || name.dirty) && name.invalid">

<li \*ngFor="let error of getValidationMessages(name)">

{{error}}

</li>

</ul>

</div>

<button class="btn btn-primary mt-2" type="submit">

Créer

</button>

</form>

</div>

Illustration 12-21Exécution de la validation à l'échelle du formulaire dans le fichier template.html du dossier src/app

L'élément form définit maintenant une variable de référence appelée form, qui a été assignée à ngForm. C'est ainsi que la directive ngForm permet d'accéder à ses fonctionnalités, par un processus que je décrirai au chapitre 13. Pour l'instant, il est important de savoir que les informations de validation de l'ensemble du formulaire sont accessibles via la variable de référence form.

La liste modifie également l'expression de la liaison ngSubmit afin qu'elle appelle la méthode submitForm définie par le contrôleur, en passant par la variable template, comme ceci :

...

<form ngForm="productForm" #form="ngForm" (ngSubmit)="submitForm(form)">

...

C'est cet objet qui est reçu comme argument de la méthode submitForm et qui est utilisé pour vérifier l'état de validation du formulaire et pour réinitialiser le formulaire afin qu'il puisse être utilisé à nouveau.

La liste 12-21 ajoute également un élément div qui utilise la propriété formSubmitted du composant ainsi que la propriété valid (fournie par la variable de modèle de formulaire) pour afficher un message d'avertissement lorsque le formulaire contient des données non valides, mais seulement après que le formulaire a été soumis.

En outre, la liaison ngIf a été mise à jour pour afficher les messages de validation au niveau des champs de manière à ce qu'ils s'affichent lorsque le formulaire a été soumis, même si l'élément lui-même n'a pas été édité. Le résultat est un résumé de validation qui n'est affiché que lorsque l'utilisateur soumet le formulaire avec des données invalides, comme l'illustre la figure 12-11.

Figure 12-11Affichage d'un message récapitulatif de validation

Affichage de messages de validation récapitulatifs

Dans un formulaire complexe, il peut être utile de fournir à l'utilisateur un résumé de toutes les erreurs de validation à résoudre. L'objet NgForm assigné à la variable de référence du modèle de formulaire permet d'accéder aux éléments individuels par le biais d'une propriété nommée controls. Cette propriété renvoie un objet qui possède des propriétés pour chacun des éléments individuels du formulaire. Par exemple, il existe une propriété name qui représente l'élément input dans l'exemple, à laquelle est assigné un objet qui représente cet élément et définit les mêmes propriétés de validation que celles qui sont disponibles pour les éléments individuels. Dans le Listing 12-22, j'ai ajouté une méthode au composant qui reçoit l'objet assigné aux variables de référence du modèle de l'élément de formulaire et utilise sa propriété controls pour générer une liste de messages d'erreur pour l'ensemble du formulaire.

import { Component } from "@angular/core" ;

import { Model } from "./repository.model" ;

import { Product } from "./product.model" ;

import { NgModel, ValidationErrors, NgForm } from "@angular/forms" ;

@Component({

selector : "app",

templateUrl : "template.html"

})

export class ProductComponent {

modèle : Modèle = nouveau Modèle() ;

// ...autres méthodes omises par souci de concision...

getFormValidationMessages(form : NgForm) : string[] {

let messages : string[] = [] ;

Object.keys(form.controls).forEach(k => {

this.getMessages(form.controls[k].errors, k)

.forEach(m => messages.push(m)) ;

}) ;

return messages ;

}

}

Listing 12-22Génération de messages de validation à l'échelle du formulaire dans le fichier component.ts du dossier src/app

La méthode getFormValidationMessages construit sa liste de messages en appelant la méthode getMessages pour chaque contrôle du formulaire. La méthode Object.keys crée un tableau à partir des propriétés définies par l'objet renvoyé par la propriété controls, qui est énuméré à l'aide de la méthode forEach.

Dans le Listing 12-23, j'ai utilisé cette méthode pour inclure les messages individuels en haut du formulaire, qui seront visibles une fois que l'utilisateur aura cliqué sur le bouton Créer.

<div class="p-2">

<form #form="ngForm" (ngSubmit)="submitForm(form)">

<div class="bg-danger text-white p-2 mb-2"

\*ngIf="formSubmitted && form.invalid">

Il y a des problèmes avec le formulaire

<ul>

<li \*ngFor="let error of getFormValidationMessages(form)">

{{erreur}}

</li>

</ul>

</div>

<div class="form-group">

<label>Nom</label>

<input class="form-control"

name="nom"

[(ngModel)]="nouveauProduit.nom"

#nom="ngModel"

obligatoire

minlength="5"

pattern="^[A-Za-z ]+$" />

<ul class="text-danger list-unstyled mt-1"

\*ngIf="(formSubmitted || name.dirty) && name.invalid">

<li \*ngFor="let error of getValidationMessages(name)">

{{error}}

</li>

</ul>

</div>

<button class="btn btn-primary mt-2" type="submit">

Créer

</button>

</form>

</div>

Liste 12-23Affichage des messages de validation à l'échelle du formulaire dans le fichier template.html du dossier src/app

Le résultat est que les messages de validation sont affichés à côté de l'élément de saisie et rassemblés en haut du formulaire une fois que celui-ci a été soumis, comme le montre la figure 12-12.

Figure 12-12Affichage d'un résumé général de la validation

Désactivation du bouton Soumettre

L'étape suivante consiste à désactiver le bouton une fois que l'utilisateur a soumis le formulaire, en l'empêchant de cliquer à nouveau sur le bouton tant que toutes les erreurs de validation n'ont pas été résolues. Il s'agit d'une technique couramment utilisée, même si elle n'a que peu d'incidence sur l'application de l'exemple, qui n'acceptera pas les données du formulaire tant qu'il contiendra des valeurs non valides, mais qui renforce utilement l'idée que l'utilisateur ne peut pas continuer tant que les problèmes de validation n'ont pas été résolus. Dans le Listing 12-24, j'ai utilisé la liaison de propriété sur l'élément button.

<div class="p-2">

<form #form="ngForm" (ngSubmit)="submitForm(form)">

<div class="bg-danger text-white p-2 mb-2"

\*ngIf="formSubmitted && form.invalid">

Il y a des problèmes avec le formulaire

<ul>

<li \*ngFor="let error of getFormValidationMessages(form)">

{{erreur}}

</li>

</ul>

</div>

<div class="form-group">

<label>Nom</label>

<input class="form-control"

name="nom"

[(ngModel)]="nouveauProduit.nom"

#nom="ngModel"

obligatoire

minlength="5"

pattern="^[A-Za-z ]+$" />

<ul class="text-danger list-unstyled mt-1"

\*ngIf="(formSubmitted || name.dirty) && name.invalid">

<li \*ngFor="let error of getValidationMessages(name)">

{{error}}

</li>

</ul>

</div>

<button class="btn btn-primary mt-2" type="submit"

[disabled]="formSubmitted && form.invalid"

[class.btn-secondary]="formSubmitted && form.invalid">

Créer

</button>

</form>

</div>

Liste 12-24Désactivation du bouton dans le fichier template.html du dossier src/app

Pour plus d'insistance, j'ai utilisé la liaison de classe pour ajouter l'élément bouton à la classe btn-secondaire lorsque le formulaire a été soumis et que les données ne sont pas valides. Cette classe applique un style CSS Bootstrap, comme le montre la figure 12-13.

Figure 12-13Désactiver le bouton d'envoi

Compléter le formulaire

Maintenant que les fonctions de validation sont terminées, je peux compléter le formulaire. Le Listing 12-25 rétablit les éléments de saisie pour les champs category et price, que j'ai supprimés plus tôt dans le chapitre. J'ai également supprimé les messages de validation pour l'élément name afin que seuls les messages d'erreur du formulaire soient affichés.

<div class="p-2">

<form #form="ngForm" (ngSubmit)="submitForm(form)">

<div class="bg-danger text-white p-2 mb-2"

\*ngIf="formSubmitted && form.invalid">

Il y a des problèmes avec le formulaire

<ul>

<li \*ngFor="let error of getFormValidationMessages(form)">

{{erreur}}

</li>

</ul>

</div>

<div class="form-group">

<label>Nom</label>

<input class="form-control"

name="nom"

[(ngModel)]="nouveauProduit.nom"

#nom="ngModel"

obligatoire

minlength="5"

pattern="^[A-Za-z ]+$" />

</div>

<div class="form-group">

<label>Catégorie</label>

<input class="form-control" name="category"

[(ngModel)]="nouveauProduit.catégorie" obligatoire />

</div>

<div class="form-group">

<label>Prix</label>

<input class="form-control" name="price"

[(ngModel)]="nouveauProduit.prix" required type="nombre"/>

</div>

<button class="btn btn-primary mt-2" type="submit"

[disabled]="formSubmitted && form.invalid"

[class.btn-secondary]="formSubmitted && form.invalid">

Créer

</button>

</form>

</div>

Liste 12-25Ajouter des éléments de formulaire dans le fichier template.html du dossier src/app

La dernière modification consiste à ajuster les sélecteurs des styles CSS qui indiquent les éléments d'entrée valides et non valides, comme le montre l'illustration 12-26.

html, body { height : 100% ; }

body { margin : 0 ; font-family : Roboto, "Helvetica Neue", sans-serif ; }

form.ng-submitted input.ng-invalid { border : 2px solid #ff0000 }

form.ng-submitted input.ng-valid { border : 2px solid #6bc502 }

Liste 12-26Ajustement des sélecteurs CSS dans le fichier styles.css du dossier src

En plus des classes décrites dans le tableau 12-5, Angular ajoute les éléments de formulaire à la classe ng-submitted lorsqu'ils ont été soumis. Cela me permet de sélectionner les éléments qui sont invalides une fois que le formulaire a été soumis, que l'utilisateur ait ou non édité les éléments.

Enregistrez les modifications et cliquez sur le bouton Créer ; vous verrez apparaître les messages de validation et les styles CSS illustrés à la figure 12-14. Au fur et à mesure que vous corrigez les erreurs de validation, les éléments de saisie deviennent verts et vous pouvez soumettre le formulaire lorsqu'il n'y a plus d'erreurs de validation.

Figure 12-14Finition du formulaire

Résumé

Dans ce chapitre, j'ai présenté la manière dont Angular prend en charge l'interaction avec l'utilisateur à l'aide d'événements et de formulaires. J'ai expliqué comment créer des liaisons d'événements, comment créer des liaisons bidirectionnelles et comment elles peuvent être **simplifiées à l'aide de la directive ngModel**. J'ai également décrit le support qu'Angular fournit pour la gestion et la validation des formulaires HTML. Dans le chapitre suivant, j'explique comment créer des directives personnalisées.

table des matières

recherche

Paramètres

file d'attente

retour