GESTIÓN DE CALIDAD SOFTWARE

Fundamentos 2ª Parte: Formularios y Navegación

Autor: Santiago Meliá. DLSI. UA

Índice

- Formularios
  - Introducción
  - Reactive forms
  - Form Control
  - Form Group
  - Validación con Reactive Forms
  - Form Array
- Routing y Navegación
  - Angular Routing
  - Tipos de enrutamiento y navegación
  - Lazy Routing
  - Enlaces dinámicos
- Ejercicio 3: Navegación entre páginas y formulario de Login

Formularios. Introducción

- Manejar la entrada de datos mediante forms es una parte fundamental en la mayoría de aplicaciones

- Aplicaciones usan los forms para logarse, actualizar su perfil o entrar información sensible, y en general, para introducir cualquier tipo de dato

- Angular proporciona dos tipos de aproximaciones para manejar la entrada de datos:
  - Reactive forms
  - Template-driven forms

Formularios. Introducción

- Ambas aproximaciones capturan los eventos de entrada del usuario desde la vista, los validan y crean modelo para actualizar
- Reactive y template driven procesan y manejan los datos del formulario de forma distinta
  - **Reactive Forms** – son más robustos, escalables, reusables y testables. Si los formularios son la parte fundamental de tu aplicación, es recomendable usarlos
  - **Template-driven Forms** – son útiles para añadir un formulario simple a una app, tal como darse de alta a una lista de correo. Su principal ventaja es que son más fáciles de usar

Elementos comunes

- Tanto Reactive como template-driven forms comparten una serie de elementos comunes:
  - **FormControl** – monitorea el valor y el estado de validación de un control del formulario individual
  - **FormGroup** – monitorea los valores y estado de una colección de controles del formulario
  - **FormArray-** monitorea los valores y el estado de un array de controles del formulario
  - **ControlValueAccessor** – crea un puente entre el FormControl de angular y los elementos nativos del DOM

Reactive Forms

- Proveen una aproximación orientada a modelos para manejar los formularios de entrada cuyos valores cambian en el tiempo
- Usan una aproximación explícita para manejar el estado de un formulario en un momento dado
- Cada cambio al estado del formulario devuelve un nuevo estado, lo que mantiene la integridad del modelo entre cada cambio
- Los reactive forms están basados en los Observables streams, donde los inputs del formulario y los valores son proporcionados como streams de valores de entrada que pueden ser accedidos síncronamente

Reactive Forms

- Reactive forms provee una forma sencilla para realizar las pruebas porque se asegura que los datos son consistentes y predecibles en cada petición
- Cualquier consumidor de los streams tienen acceso para manipular los datos del formulario de una forma segura mediante los operadores de los observables
- El primer paso es importar el módulo ReactiveFormsModule en el módulo donde vayamos a trabajar

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

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

Importación de un FormControl

- La clase FormControl es el elemento básico cuando vamos a usar los reactive forms
- El siguiente paso consiste en importar la clase FormControl en nuestro componente o página y crear una nueva instancia del form control para salvar como una propiedad de clase
- El constructor recibe el valor inicial del FormControl (en este caso cadena vacía)
- Creando estos controles, tienes acceso a escuchar, actualizar y validar el estado del campo de entrada

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

@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css']
})
export class NameEditorComponent {
  name = new FormControl('');
}
```

Registrar el control en el template

- Una vez creado el control en el componente, se debe asociar dicho control a un elemento en el template
- FormControl se puede enlazar a los elementos HTML tipo *input* o *select*
- Para ello se usa la directiva [formControl] sobre el elemento (contenida en ReactiveFormsModule)
- Queda enlazado el form control con el elemento del DOM input llamado “name”.
- Así la vista refleja los cambios del modelo y el modelo los cambios de la vista (doble binding)

```
<label>
  Name:
  <input type="text" [formControl]="name">
</label>
```

Manejando los valores del control

- Los reactive forms te proporcionan acceso al estado y al valor del form control en un momento dado
- Se pueden manipular el estado y su valor mediante la clase del componente o en la propia plantilla
- Existen dos modos para mostrar los valores del control:
  - Mediante el observable *valueChanges* que permite escuchar los cambios en el valor del formControl
  - Con la propiedad *value* que te da un snapshot del valor actual. El valor mostrado cambia cuando se actualiza el formControl asociado

```
<p>
  Value: {{ name.value }}
</p>
```

Cambiando un valor del formControl

- Los reactive forms proporcionan métodos para cambiar el valor del control programáticamente, te permite actualizar el valor sin la interacción del usuario
- Un formControl tiene el método *setValue()* que actualiza el valor y se realiza su validación
- Ejemplo: Cuando se pulsa el botón se actualiza el valor el formControl

```
updateName() {
  this.name.setValue('Nancy');
}
```

```
<p>
  <button (click)="updateName()">Update Name</button>
</p>
```

Agrupando controles del formulario

- Una instancia de FormGroup monitorea el estado de un grupo de controles del formulario. A cada control de un FormGroup se accede por su nombre
- Para inicializar un formGroup se crea una propiedad en el componente de tipo FromGroup, a cuyo constructor le pasamos un conjunto de FormControl con su nombre
- Los controles individuales así son agrupados en un grupo. Una instancia de FormGroup provee su valor como un objeto formado por los valores de los controles que contiene. Pero contiene las mismas propiedades de un control individual (value y untouched)

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });
}
```

Asociando un FormGroup a una vista

- En un FormGroup si un control cambia, el control grupal emite un nuevo estado o un cambio de valor
- El modelo del FormGroup es enlazado con el elemento form mediante la directiva [formGroup]
- Para cada control individual se hace uso de la directiva formControlName que enlaza cada input individual a cada formControl dentro del FormGroup

```
<form [formGroup]="profileForm">

  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>

</form>
```

Enviar los datos de un FormGroup

- La directiva FormGroup escucha el evento submit del formulario y emite un evento *ngSubmit* que se enlaza a una función callback del componente
- Llamamos a la función callback onSubmit() del componente

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">
```

- Muestra en la consola el valor del formulario

```
onSubmit() {
  // TODO: Use EventEmitter with form value
  console.warn(this.profileForm.value);
}
```

- El evento submit es emitido por la etiqueta form de DOM. Para ello, se define un botón de tipo submit

```
<button type="submit" [disabled]="!profileForm.valid">Submit</button>
```

- La directiva disabled deshabilita el botón cuando el modelo es inválido

Validación de formularios

- El Form Validation es usado para validar la entrada de un usuario y asegurar que es completa y correcta
- Reactive Forms incluyen un conjunto de funciones de validación para casos comunes
- Cada función recibe el control sobre el cual validar y devuelve un objeto error o un null (si va bien) basado en la comprobación de validación
- Pueden utilizarse las funciones de validación que proporciona angular o implementarnos las nuestras
- Las funciones de validación pueden ser síncronas (devuelve inmediatamente el error o null) o asíncronas (emite un observable que posteriormente emite el conjunto de errores)

Funciones de validación predefinidas

- Las funciones compose y composeAsync permiten componer múltiples validadores en una sola función

```
class Validators {
  static min(min: number): ValidatorFn
  static max(max: number): ValidatorFn
  static required(control: AbstractControl): ValidationErrors | null
  static requiredTrue(control: AbstractControl): ValidationErrors | null
  static email(control: AbstractControl): ValidationErrors | null
  static minLength(minLength: number): ValidatorFn
  static maxLength(maxLength: number): ValidatorFn
  static pattern(pattern: string | RegExp): ValidatorFn
  static nullValidator(control: AbstractControl): ValidationErrors | null
  static compose(validators: ValidatorFn[]): ValidatorFn | null
  static composeAsync(validators: AsyncValidatorFn[]): AsyncValidatorFn | null
}
```

Ejemplo de validación

- Se indica el nombre del control, a continuación el conjunto de validaciones en un array
- Se ha indicado que el campo name es requerido, su longitud mínima es 4, y se invoca a un validador custom para comprobar que el nombre es permitido

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    'name': new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.
    ]),
    'alterEgo': new FormControl(this.hero.alterEgo),
    'power': new FormControl(this.hero.power, Validators.required)
  });

}
```

Ejemplo de validación

- Se hace uso de las clases CSS para mostrar los errores de la validación
- El atributo required no es necesario para la funcionalidad, solo por motivos estéticos

```
<input id="name" class="form-control"
      formControlName="name" required >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
    class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>
</div>
```

Generador controles form con FormBuilder

- Crear instancias FormControl manualmente es algo repetitivo cuando trabajas con múltiples formularios
- El servicio FormBuilder proporciona los métodos necesario para crear controles
- Para utilizarlo debemos importar el FormBuilder, e inyectarlo al componente que contiene el formulario:

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

```
constructor(private fb: FormBuilder) { }
```

Creando controlores con formBuider

- El servicio FormBuilder tiene tres métodos de factoría para crear instancias en el componente de tipo form controls (control()), form groups (group()) y arrays (array())

```
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

  constructor(private fb: FormBuilder) { }
}
```

```
profileForm = new FormGroup({
  firstName: new FormControl(''),
  lastName: new FormControl(''),
  address: new FormGroup({
    street: new FormControl(''),
    city: new FormControl(''),
    state: new FormControl(''),
    zip: new FormControl('')
  })
});
```

Uso de FormArray para los formularios

- Es una alternativa a FormGroup para manejar cualquier número de controles sin su nombre
- Es la mejor opción si no se conoce el número de controles por adelantado
- Se hace uso del FromBuilder.array() para definir el array y del FormBuilder.control para rellenar el array con los controles iniciales
- Por lo tanto debemos importar tanto el FormBuilder como el FormArray

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

Definiendo un control FormArray

- Se puede inicializar un form array con cualquier número de controles, de cero a muchos.
- Se debe crear una propiedad al formGroup para contener el form array (p.e.aliases)
- Ejemplo, iniciamos el form array con un único control

```
profileForm = this.fb.group({
  firstName: ['', Validators.required],
  lastName: [''],
  address: this.fb.group({
    street: [''],
    city: [''],
    state: [''],
    zip: ['']
  }),
  aliases: this.fb.array([
    this.fb.control('')
  ])
});
```

Accediendo al control FormArray

- Usamos un getter para recuperar el FormArray

```
get aliases() {
  return this.profileForm.get('aliases') as FormArray;
}
```

- Definimos un método para insertar dinámicamente un control al form array. El método FormArray.push incluye un control nuevo en el array

```
addAlias() {
  this.aliases.push(this.fb.control(''));
}
```

Mostrando el form array en la vista

- Se utiliza la directiva formArrayName que enlaza la comunicación del form array hasta el template
- ngFor muestra todos los controles del form array. Al no tener nombre los controles, se les asigna el índice de la iteración (i) como nombre

```
<div formArrayName="aliases">
  <h3>Aliases</h3> <button (click)="addAlias()">Add Alias</button>

  <div *ngFor="let address of aliases.controls; let i=index">
    <!-- The repeated alias template -->
    <label>
      Alias:
      <input type="text" [formControlName]="i">
    </label>
  </div>
</div>
```

Navegación en Ionic 4

- Uno de los cambios más importantes de Ionic 3 a Ionic 4 es como el routing y como la navegación es manejada
- En el pasado, ionic usaba su propio push-pop NavController, pero hay en la versión actual ha dado un paso decidido a usar el Angular Router
- Es un movimiento inteligente del equipo de Ionic debido a que el router de Angular es mucho más potente. Eso sí, tiene una mayor curva de aprendizaje que la forma anterior
- Todavía se mantiene el componente ion-nav pero su tendencia es a desaparecer en el futuro

Enrutamientos con Angular Route

- Se definen un conjunto de routes que se pueden cargar en uno o más módulos
- Cada route define una relación entre un path URL y el componente o módulo al que nos dirigimos
- Si es estamos en un módulo entonces, este define su propio route para indicar como se redirige al componente o página
- Hay tres tipos de enrutamientos que se pueden usar:
  - **Eager-loaded**: rutas que apunta a un solo componente
  - **Lazy-loaded**: rutas que apuntan a un módulo. Dicho módulo puede tener a su vez un conjunto de componentes
  - **Redirect**: una ruta que redirige a otra ruta

Definir enrutamiento principal de una App

- Se define una única instancia singleton del **RouterModule** en el fichero app-routing.module
- Se define un objeto **Routes** como un array de rutas. Que es pasado al módulo con el método forRoot para indicar que son las rutas raiz
- Cuando la URL cambia, el router busca el correspondiente path para determinar el componente a mostrar
- En el ejemplo se usa un **lazy routing**, ya que se está navegando a un módulo y este tiene su su propio router que nos redireccionará a el/los componente/s

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  { path: '', loadChildren: './tabs/tabs.module#TabsPageModule' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Otro ejemplos de enrutamiento

- Las tres primeras rutas redirigen al componente directamente (eager-loaded)
- La segunda ruta ‘hero/:id’ es un ejemplo de paso de parámetros en enlace dinámico
- La tercera ruta permite establecer datos a una ruta, puede ser usado para los títulos de página, migas de pan (datos estáticos)
- La cuarta ruta es el típico redireccionamiento cuando el path es vacío
- La quinta ruta es un comodín, si ninguna ruta coincide te envía a la página 404

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];
```

Router outlet

- La directiva RouterOutlet de la librería router es el placeholder o contenedor que marca el punto del template donde se van a mostrar los componentes que devuelve el enrutamiento
- Normalmente está ubicada en el app-component.html

```
<ion-app>
  <ion-router-outlet></ion-router-outlet>
</ion-app>
```

Navegación desde la plantilla

- Un enlace o un botón es la forma más sencilla de navegar a una ruta. Dichas rutas son las indicadas en la parte path de los router

```
<ion-button href="/hello">Hello</ion-button>
```

- Se debe usar la sintaxis de Angular router para los componentes que no sean ionic. Con la directiva *routerLink*

```
<a routerLink="/hello">Hello</a>
```

Navegación desde Código

- Para navegar desde el código de un componente o servicio se puede usar el servicio Router. Dicho servicio debe ser inyectado al componente o servicio que lo utilice
- Se especifica la URL a la cual navegar (normalmente un path del router) mediante el método navigateByURL

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

@Component({ ... })
export class HomePage {
  constructor(private router: Router) {}

  go() {
    this.router.navigateByUrl('/animals');
  }
}
```

Lazy routing

- Ionic apuesta por el uso la carga mediante módulos y esto influye en la propia navegación
- Cuando navegamos con Lazy routing navegamos a un módulo feature que contiene un conjunto de componentes
- Dicho modulo contiene su propio Route con la lista de path componentes (eager route) al cual apunta

Este es el route del app-routing.module.ts que contiene el path hasta el módulo

```
const routes: Routes = [
  { path: 'about', loadChildren: './about/about.module#AboutPageModule' },
];
```

Y este route pertenece al modulo about.module.ts, y su ruta por defecto ’’ lleva al componente AboutPage

```
const routes: Routes = [
  { path: '', component: AboutPage },
];
```

Lazy routing

- La lista de rutas que contienen los módulos deben incluirse en el RouteModule para que formen parte de ellas
- Para ello se utilizará el método forChild
- Las rutas de un módulo llevarán como prefijo el path que les dirigío hasta dicho módulo
- En el ejemplo anterior llevarían el prefijo ‘about’ todos los paths indicados en el about.module

```
@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    ReactiveFormsModule,
    RouterModule.forChild(routes)
  ],
  declarations: [LoginPage]
})
export class LoginPageModule {}
```

Enlaces dinámicos

- En muchos casos se necesita navegar a una página y pasarle un valor que no se conoce de antemano
- Se define una ruta que permita recibir valores dinámicos. El segmento dinámico se prefija con ‘:’

```
const routes: Routes = [
  // Regular Route
  { path: 'items/:id', component: MyComponent },
];
```

```
<ion-button href="/items/abc">ABC</ion-button>
<ion-button href="/items/xyz">XYZ</ion-button>
```

Extraer datos de las rutas con ActivatedRoute

- Cuando se trabaja con datos dinámicos, en ocasiones se necesita extraer los parámetros de la URL
- Después de cada ciclo de vida de navegación, el router crea un árbol de objetos *ActivatedRoute* que definen el estado actual del router
- Se puede acceder al estado del router desde cualquier lugar de la aplicación usando el servicio *Router* y la propiedad *routerState*

Extraer datos de las rutas con ActivatedRoute

- Basados en el ejemplo anterior, queremos recuperar de la BBDD los items de un determinado id
- El servicio ActivatedRoute permite recuperar la información de la ruta como un objeto plano o como un observable

```
ionViewWillEnter (){

    this.cardId = this.route.snapshot.paramMap.get('cardId');
```

Navegación mediante ion-nav

- Una forma sencilla de establecer la navegación es usando el controlador de navegación **ion-nav**
- Para ello, se debe inyectar en cualquier de las páginas donde lo uses
- Si cargamos un Nav Component y hacemos push de nuevos componentes en la pila, esto no afecta a ningún router
- Se usa en aquellos casos donde necesitas una subnavegación que no se ate a URLs de la app
- No es recomendable su uso ya que el equipo de Ionic tiene previsto prescindir de esta forma de navegación próximamente

Ejercicio 3: HearthStoneApp

1. Creamos la página **card-listing** en la ruta *src/card-listing* con *ionic generate*
2. Usando el lazy routing, añadimos el path dinámico "**card-listing/:cardDeckGroup/:cardDeck** al módulo de ***card-deck*** y enviamos (loadChildren) al modulo de CardListing (ver pag. 34)
3. Borrar la nueva ruta que se crea en app.routing.ts que apunta a card-listing
4. Añadir una función generateURL en card-deck que navegue a card-listing con la dirección ***"/tabs/cards/card-listing/" + cardDeckGroup + "/" + cardDeck***. Recibiendo dos parámetros tipo string *cardDeckGroup* y *cardDeck*.
5. Añadimos al componente *card-list* una propiedad input @input () llamada ***navigateTo*** de tipo *any*
6. En el template de *card-list* dentro el control ion-item añadimos la directiva [href] y le pasamos en la derecha "***navigateTo(listname, item)"***
7. En la plantilla de la *card-deck* en la invocación al componente card-list añadimos la propiedad [**navigateTo**] y le pasamos "**generateURL**"

Ejercicio 3: HearthStoneApp (Cont. 1)

8. Al constructor del CardListingPage le inyectamos un servicio ActivatedRoute (***private route:ActivatedRoute***)
9. Definimos 2 propiedades en CardListingPage de tipo string. CardDeckGroup y CardDeck
10. Quitamos el método ngInit (y también el implements a Init) y añadimos el método ionViewWillEnter ()
11. Recuperamos los parámetros CardDeckGroup y CardDeck con el ActivatedRoute (ver pag. 36)
12. Definimos el interface Card en card.models para poder tener tipado el array de cartas que recibimos en el servicio (ver pag. 41)
13. Definimos un método en el servicio CardService llamado getCardByDeck que recupera las cartas por el mazo (ver pag. 42)
14. Definimos una propiedad cards de tipo array de Card en CardListingPage ***cards: Card[] = [];***
15. Nos suscribimos al método **getCardByDeck** en CardListingPage dentro de su método ionViewWillEnter. Antes debemos inyectar el *CardService* a la página CardListingPage. (ver pag. 42)

Ejercicio 3: HearthStoneLib (Cont. 3)

16. En la plantilla de card-listing definimos las tarjetas para mostrar la lista de Cartas en card-listing (ver pag. 43)
17. En el título de la plantilla mostramos el CardDeck mediante interpolación {{}}
18. Creamos una página llamada **login** con ionic generate en src/login
19. Tenemos que importar al módulo de login el ReactiveFormsModule
20. Debemos inyectar el **FormBuilder** para crear en el formulario de login con los campos email y password que sean requeridos. Y el email sea correcto (Ver. 22 y 17)
21. Se inyecta también el servicio Router y se define el método **goToHome** que nos llevará a la URL '*tabs/cards*' con el método *navigateByURL*
22. Definimos parte estática del formulario en la plantilla, donde definimos los dos campos de entrada y el botón de entrar. (Ver pag. 44)
23. Debemos actualizar el app-routing para que redireccione al Login. Para ello hacemos que la dirección vacía '' haga un redirect que nos lleve a la pantalla de login '/login'
24. Cambiamos el router de app-routing de tabs, poniendole en el path = 'tabs' y en tabs.router donde donde pone 'tabs' poner ahora vacío

Definimos el interface Card

- Utilizamos el tipo Card para recuperar el conjunto de cartas que nos devuelve el interfaz de HearthStone y pintarlas en Card-listing

```
export interface Card {
  text: string;
  cardId: string;
  cardSet: string;
  collectible: boolean;
  img: string;
  imgGold: string;
  name: string;

  cost: number;
  attack: number;
  health:number;
  rarity: string;
  type: string;

  dbfId: string;
  faction:string;
  playerClass: string;
  locale: string;

}
```

Método para recuperar las cartas por el mazo (Deck)

- El método está ubicado en el servicio CardService y devuelve un Observable al que nos suscribimos

```
public getCardByDeck(cardDeckGroup:string, cardDeck:string): Observable<Card[]>{
  return this.http.get<Card[]>(this.HS_API_URL+'/cards/'+cardDeckGroup+'/'+cardDeck, {headers: this.headers});

}
```

- Nos suscribimos al método getCardByDeck desde la clase CardListingPage, recibimos el array de cartas y se lo asignamos a la propiedad cards de la clase

```
this.cardService.getCardByDeck (this.cardDeckGroup, this.cardDeck).subscribe (
  (cards:Card[]) => {
    this.cards = cards;
 });
```

Componente card-listing

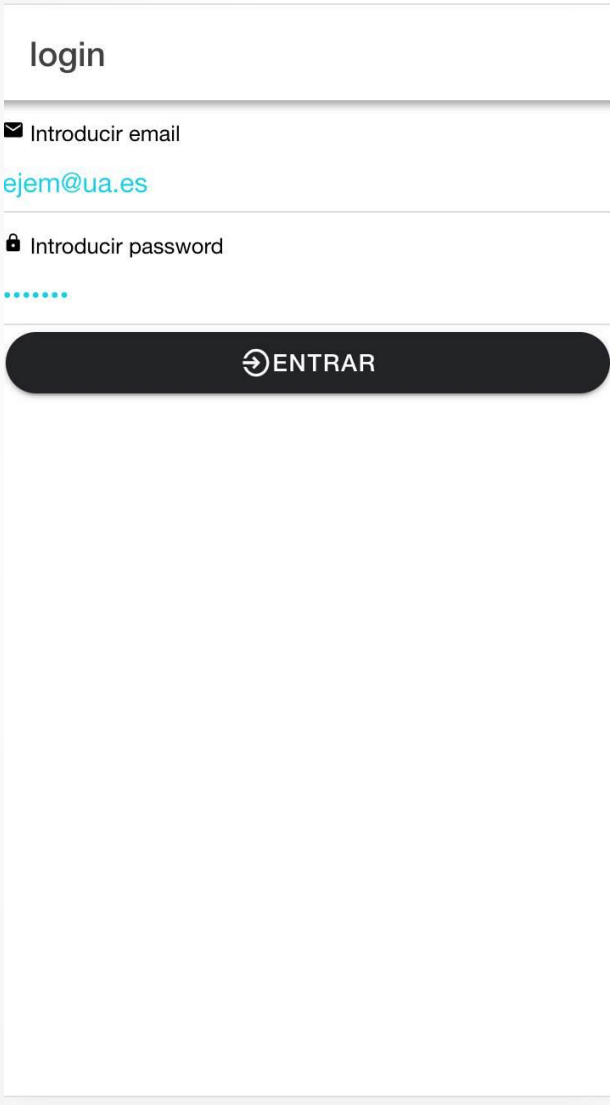
- La vista del componente se enlaza con sus dos propiedades

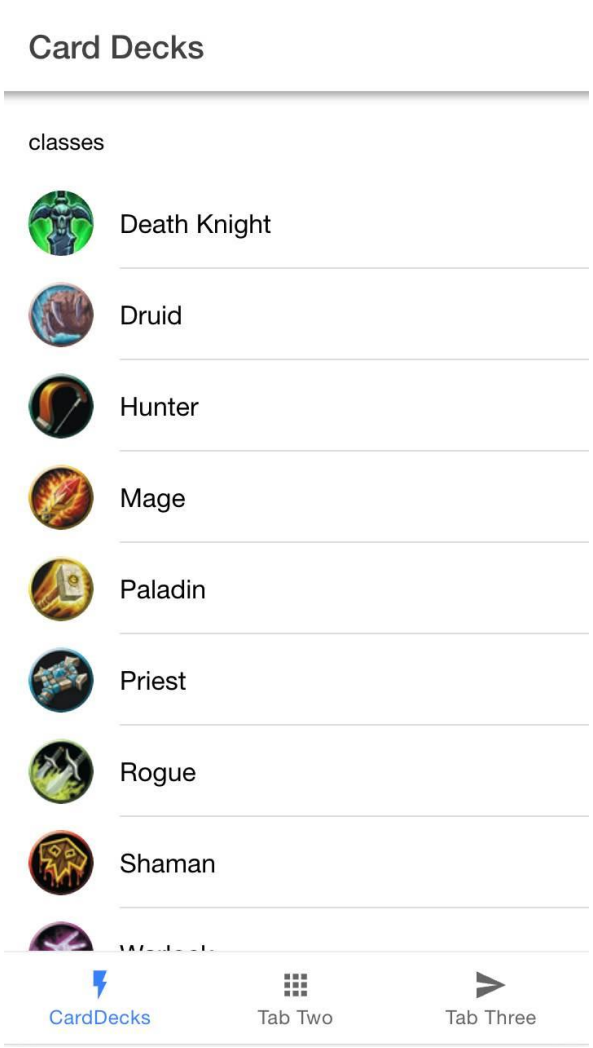
```
<ion-list *ngIf="cards.length > 0">
  <ion-card *ngFor="let card of cards">
    <ion-card-header text-wrap>
      <ion-card-subtitle>
        {{card.cardSet}}
      </ion-card-subtitle>
      <ion-card-title>
        {{card.name}}
      </ion-card-title>
    </ion-card-header>
    <ion-card-content>
      <div [innerHTML]="card?.text"></div>
      <ion-button size="medium" expand='full' [href]="">
        See details
      </ion-button>
    </ion-card-content>
  </ion-card>
</ion-list>
```

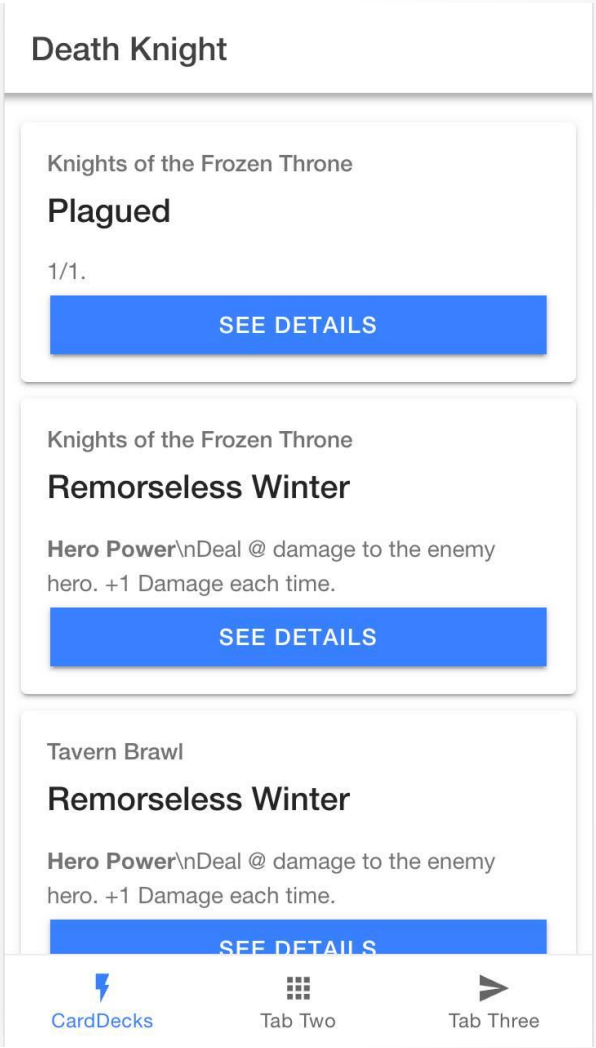
Plantilla de la pantalla login

```
<form [formGroup]="onLoginForm" class="list-form">
  <ion-item no-padding class="animated fadeInUp">
    <ion-label position="floating">
      <ion-icon name="mail" item-start></ion-icon>
      Introducir email
    </ion-label>
    <ion-input color="secondary" type="email" formControlName="email"></ion-input>
  </ion-item>
  <p ion-text class="text08" *ngIf="onLoginForm.get('email').touched && onLoginForm.get('email').hasError('required')">
    <ion-text color="warning">
      Email incorrecto
    </ion-text>
  </p>

  <ion-item no-padding class="animated fadeInUp">
    <ion-label position="floating">
      <ion-icon name="lock" item-start></ion-icon>
      Introducir password
    </ion-label>
    <ion-input color="secondary" type="password" formControlName="password"></ion-input>
  </ion-item>
  <p ion-text color="warning" class="text08" *ngIf="onLoginForm.get('password').touched && onLoginForm.get('password').hasError('required')">
    <ion-text color="warning">
      Password incorrecto
    </ion-text>
    </p>
</form>
<div>
  <ion-button icon-left size="medium" expand="full" shape="round" color="dark" (click)="goToHome()" [disabled]="!onLoginForm.valid" tappable>
    <ion-icon name="log-in"></ion-icon>
    Entrar
  </ion-button>

</div>
```

Solución final







Documentación

- Ionic:
  - https://ionicframework.com/docs
- Angular:
  - https://angular.io/docs