GESTIÓN DE CALIDAD SOFTWARE

Fundamentos 1ª Parte: Componentes, Servicios y Observables

Autor: Santiago Meliá. DLSI. UA

Índice

- Interacción entre los componentes
  - Paso de padre a hijo con input binding
  - Paso de hijo a padre mediante EventEmitter
  - Padre interactúa con el hijo mediante variable
  - Padre interactúa con el hijo usando @ViewChild
- Servicios
  - Definición
  - Inyección de Dependencia y Service Provider
  - Observables & RxJS
  - HTTPClient. Comunicación Cliente-Servidor
- Ejercicio: Recuperación de datos desde el API de HearthStone

Interacción entre Componentes

- Como se ha indicado, el elemento de construcción de Ionic 4 son los componentes
- Las páginas son componentes que pueden contener otros componentes a su vez y así sucesivamente
- Cada componente representa una parte de la página que es reutilizable en toda la App
- Se debe así establecer una interacción entre dichos componentes que requiere el paso de datos del componente contendedor (o padre) al componente contenido (o hijo) y viceversa

Paso del padre al hijo mediante el input binding

- Para ello se usa el decorador @input que pasa los datos entre el componente padre hacia el componente hijo
- En la plantilla del padre se añade la etiqueta del hijo pasándole los datos como atributos con corchetes [] usando binding
- El decorador @input define una propiedad como entrada de datos o setteable del hijo

```
@Component({
  selector: 'app-hero-parent',
  template: `
    <h2>{{master}} controls {{heroes.length}} heroes</h2>
    <app-hero-child *ngFor="let hero of heroes"
      [hero]="hero"
      [master]="master">
    </app-hero-child>
  `
})
```

```
export class HeroChildComponent {
  @Input() hero: Hero;
  @Input('master') masterName: string;
}
```

Paso del padre al hijo mediante el input binding

- Puede interceptarse el valor enviado por el padre y modificarse en el hijo con un setter

```
@Input()
set name(name: string) {
  this._name = (name && name.trim()) || '<no name set>';
}
```

- También se puede detectar cualquier cambio en el valor enviado del padre y actuar en el hijo con el método ngOnChanges()

```
export class VersionChildComponent implements OnChanges {
  @Input() major: number;
  @Input() minor: number;
  changeLog: string[] = [];

  ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
    let log: string[] = [];
    for (let propName in changes) {
```

Paso de hijo a padre mediante un evento EventEmitter

- En ocasiones, es necesario que el hijo le envié información al padre. P.e. El hijo tiene una entrada del usuario que queremos mostrar en el componente padre
- El hijo expone una propiedad EventEmitter la cual emite eventos cuando algo sucede
- El padre se bindea a dicha propiedad y reacciona a dichos eventos
- La propiedad EventEmitter es una propiedad de salida en el hijo indicada con el decorador @output

Paso de hijo a padre mediante un evento EventEmitter

- El hijo define la propiedad voted que emite un evento con un valor boolean

```
@Component({
  selector: 'app-voter',
  template: `
    <h4>{{name}}</h4>
    <button (click)="vote(true)"  [disabled]="didVote">Agree</button>
    <button (click)="vote(false)" [disabled]="didVote">Disagree</button>
  `
})
export class VoterComponent {
  @Input()  name: string;
  @Output() voted = new EventEmitter<boolean>();
  didVote = false;

  vote(agreed: boolean) {
    this.voted.emit(agreed);
    this.didVote = true;
  }
}
```

- El padre se enlaza a la propiedad (voted) y recibe el valor del hijo en la variable $event que le pasa a su vez a su propio método onVoted

```
@Component({
  selector: 'app-vote-taker',
  template: `
    <h2>Should mankind colonize the Universe?</h2>
    <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
    <app-voter *ngFor="let voter of voters"
      [name]="voter"
      (voted)="onVoted($event)">
    </app-voter>
  `
})
export class VoteTakerComponent {
  agreed = 0;
  disagreed = 0;
  voters = ['Mr. IQ', 'Ms. Universe', 'Bombasto'];

  onVoted(agreed: boolean) {
    agreed ? this.agreed++ : this.disagreed++;
  }
}
```

Padre interactúa con propiedades del hijo mediante una variable

- En ocasiones queremos que el componente padre interactue con las propiedades y los métodos del hijo directamente
- Para ello podemos incluir en la etiqueta que el padre contiene del hijo, una variable con el símbolo #
- Proporcionará la posibilidad de enlazarse con las propiedades del hijo e invocar a sus métodos solo desde la plantilla del padre, no desde el código del componente padre

Padre interactúa con propiedades del hijo mediante una variable

- En el ejemplo en la etiqueta del hijo (App-countdown-timer. Se añade una variable #timer. A partir de ahí en la plantilla se accede mediante dicha variable a las propiedades del hijo p.e timer.start()
- No hace falta el uso de @input ni @output en las propiedades del hijo

```
@Component({
  selector: 'app-countdown-parent-lv',
  template: `
  <h3>Countdown to Liftoff (via local variable)</h3>
  <button (click)="timer.start()">Start</button>
  <button (click)="timer.stop()">Stop</button>
  <div class="seconds">{{timer.seconds}}</div>
  <app-countdown-timer #timer></app-countdown-timer>
  `,
  styleUrls: ['../assets/demo.css']
})
export class CountdownLocalVarParentComponent { }
```

Padre interactúa con las propiedades del hijo mediante @ViewChild()

- Mayor integración entre dos componentes se realiza mediante el uso de @ViewChild
- Con ella no se invoca desde la plantilla, sino que se hace desde el código del componente padre
- Se define una propiedad del tipo del componente hijo, con el decorador @ViewChild
- Desde la clase del componente padre, se accede mediante dicha propiedad @ViewChild del hijo a todas sus métodos y propiedades

Padre interactúa con las propiedades del hijo mediante @ViewChild()

- El componente timer no está disponible hasta se muestra la vista del padre y se lanza el método After

```
export class CountdownViewChildParentComponent implements AfterViewInit {

  @ViewChild(CountdownTimerComponent)
  private timerComponent: CountdownTimerComponent;

  seconds() { return 0; }

  ngAfterViewInit() {
    // Redefine `seconds()` to get from the `CountdownTimerComponent.secon
    // but wait a tick first to avoid one-time devMode
    // unidirectional-data-flow-violation error
    setTimeout(() => this.seconds = () => this.timerComponent.seconds, 0);
  }

  start() { this.timerComponent.start(); }
  stop() { this.timerComponent.stop(); }
}
```

Introducción a Servicios

- Angular propone una separación de responsabilidades que otorga a los servicios la tarea de contener la funcionalidad no vinculada con la interfaz de usuario

- Descargan a los componentes de dichas tareas reduciendo así su acoplamiento con elementos que no sean IU, haciéndolos más ligeros y aumentando su cohesión

- Su funcionalidad está relacionada con la lógica de negocio como traer información en el servidor, guardar en un log, almacenarla en el storage de cliente, etc.

- El uso de la inyección de dependencia reduce el acoplamiento entre los componentes y los servicios

Definición de un Servicio

- Un servicio es básicamente es una clase que se define con el decorador @Inyectable
- Para poder ser utilizado se debe registrar al menos en un provider del componente, módulo o servicio donde se use

```
@Injectable({
  providedIn: 'root'
})
export class CartService {
  orderCounter = 0;
  orders: Array<any> = [];

  addtoCart(order, qtd) {
    this.orderCounter = this.orderCounter + 1;
    this.orders.push({id: this.orderCounter, order: order, qtd: qtd});
    return Promise.resolve();
  }
```

Inyección de dependencia

- Es un importante patrón de diseño [Fowler2002] en el cual se suministran los objetos a una clase en lugar de ser la clase quien los instancia. En este caso son los servicios los inyectados en los componentes

- El Framework DI de Angular provee de dichas dependencias cuando la clase se instancia

- Ayuda a reducir el acoplamiento entre componentes y el resto de la aplicación, lo que permite que sean más eficientes, mantenibles y testeables

Configurar un injector como Service Provider

- El decorador @Injectable marca que el servicio puede ser inyectado, pero Angular no puede inyectarlo a ninguna clase hasta que se configure el ***injector*** con un ***provider*** de ese servicio
- Un Provider indica a un injector como crear el servicio
- El sistema de inyección de dependencia es jerárquico y establece un árbol de inyectores paralelo al de componentes
- Se pueden realizar dicha configuración en tres sitios:
  - @Injectable () desde el propio servicio
  - @NgModule desde un módulo
  - @Component desde un componente

Estrategia 1: Proveer desde el Servicio

- El decorador @injectable del servicio tiene una opción de metadata llamada provideIn, donde puedes especificar:
  - **provideIn: root** – para indicar que dicho servicio sea proveído al RootModule (AppModule) y así accesible por toda la App
  - **provideIn: <nameModule>** - para indicar que dicho servicio se provea a un módulo en concreto
- Cuando se especifica el provider a este nivel, las herramientas de optimización pueden utilizar un tree shaking, que borra los servicios no utilizados por la App
- Esta opción es usada por defecto por los CLI's
- El tree shaking permite empaquetamientos de la App más reducidos

Ejemplo de providers desde los Servicios

- Servicio inyectable en toda la App

```
@Injectable({
  providedIn: 'root',
})
export class HeroService {
  constructor() { }
}
```

- Inyectable en aquellas clases que importen Module (HeroModule)

```
@Injectable({
  // we declare that this service should be created
  // by any injector that includes HeroModule.
  providedIn: HeroModule,
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

Estrategia 2: Proveer desde el módulo

- Se pueden especificar los providers en el meta del @NgModule
- O bien con el decorador del módulo @injectable y la opción providedIn
- Es recomendable usar esta opción si es un feature module donde se usa lazy loading, proveyendo solo la inyección a las clases que se crean en dicho módulo
- Si se usa la opción **providedIn:MyLazyloadModule**, el provider es quitado en tiempo de compilación, si no es usado en otra parte de la App

Ejemplo de provider en el @NgModule

- El Servicio CardService puede usarse dentro del módulo CardPageModule

```
@NgModule ({
  imports:[
    IonicModule,
    CommonModule,
    HttpClientModule,
    RouterModule.forChild(routes)
  ],
  providers: [
    CardService
  ],
  declarations:[
      CardDeckPage,
      CardListComponent
  ]
})
export class CardPageModule{}
```

Estrategia 3: Proveer desde un componente

- En el provider a nivel de componente, Angular solo permite que el servicio sea usado por dicho componente o uno de sus descendientes
- Cada instancia de componente tiene su propia instancia del servicio
- Cuando el componente se destruye también lo hace el servicio

```
@Component({
  selector: 'app-heroes',
  providers: [ HeroService ],
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

Observables

- Basados en la implementación del Patrón Observer de [Gamma, 1995]
- Permite el paso de mensajes entre los publicadores (publishers) y los suscriptores (subscribers) en una app @ionic/angular
- Son declarativos, definen una función publicando sus valores, pero no es ejecutada hasta que un consumer se suscribe
- El consumidor suscrito recibe notificaciones hasta que la función se completa o hasta que se da de baja

Observables

- Un observable puede enviar colecciones de valores de cualquier tipo (literales, mensajes o eventos) depende del contexto
- El API para recibir valores es la misma, ya sean enviados síncronamente o asíncronamente, debido a que la configuración y lógica es manejada por el observable
- Para crear un observable y empezar a recibir notificaciones, se debe llamar al método subscribe pasándole un objeto que actúa de observer

Observables

- Se crea un observable, que comenzará a escuchar las actualizaciones de geolocalización cuando un cliente se suscriba
- Devuelve un objeto Subscription que tiene el método unsubscribe para dejar de recibir notificaciones

```
// Create an Observable that will start listening to geolocation updates
// when a consumer subscribes.
const locations = new Observable((observer) => {
  // Get the next and error callbacks. These will be passed in when
  // the consumer subscribes.
  const {next, error} = observer;
  let watchId;

  // Simple geolocation API check provides values to publish
  if ('geolocation' in navigator) {
    watchId = navigator.geolocation.watchPosition(next, error);
  } else {
    error('Geolocation not available');
  }

  // When the consumer unsubscribes, clean up data ready for next subscription.
  return {unsubscribe() { navigator.geolocation.clearWatch(watchId); }};
});
```

Observables

- El subscriptor invoca al método subscribe pasándole un objeto que implementa el interface Observer
- El interface Observer define tres métodos callback para manejar tres tipos de notificaciones que el observable puede enviar

```
// Call subscribe() to start listening for updates.
const locationsSubscription = locations.subscribe({
  next(position) { console.log('Current Position: ', position); },
  error(msg) { console.log('Error Getting Location: ', msg); }
});

// Stop listening for location after 10 seconds
setTimeout(() => { locationsSubscription.unsubscribe(); }, 10000);
```

Tipos de métodos callback del Observable

- **next** (requerido) - es un manejador para cada valor enviado. Invocado cero o más veces después de que la ejecución del observable se inicie
- **error** (opcional) – es un manejador para una notificación de error. Un error detiene la ejecución de una instancia de observable
- **complete** (optional) – un manejador para la notificación de ejecución completada. Los valores con retraso son enviados al manejador next después de que la ejecución ha sido completada

La librería RxJS

- RxJS (Reactive Extensions for JavaScript) es una librería para programación reactiva usando observables que hace más fácil componer código asíncrono
- La programación reactiva es un paradigma enfocado al trabajo con flujos de datos finitos o infinitos de forma asíncrona
- La librería provee la implementación del tipo Observable, la cual es necesaria hasta que forme parte de JavaScript o que los navegadores lo soporten
- RxJS también provee clases utilidad para crear y trabajar con Observables

La librería RxJS

- Las funciones son usadas para:
  - Convertir código existente de operaciones asíncronas en observable
  - Iterar a través de los valores de un flujo o secuencia de datos
  - Mapear valores a diferentes tipos
  - Filtrar flujos de datos
  - Componer múltiples flujos de datos
- Casi todo se enfoca a optimización de los flujos de datos, mediante los *Reactive Streams*
- El uso es similar al patrón Iterator con la diferencia que nosotros no invocamos al iterador, sino que son los publicadores los responsables de notificar a los observadores por cada elemento de la secuencia

Crear Observables desde funciones

- RxJS oferta una variedad de funciones que pueden usarse para crear nuevos observables
- Por ejemplo, desde un contador con la función interval:

```
import { interval } from 'rxjs';

// Create an Observable that will publish a value on an interval
const secondsCounter = interval(1000);
// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
  console.log(`It's been ${n} seconds since subscribing!`));
```

RxJS Operators

- Los Operators son funciones que se crean en las bases de los observables para permitir la manipulación de flujos de datos o colecciones de datos
- Operaciones como Map(), Filter(), Concat() y flatMap()
- Estas operaciones permiten aplicar transformaciones sobre los flujos de datos
- El operador observa los valores emitidos del observable de origen (la secuencia), los transforma y devuelve un nuevo observable de esos valores ya transformados

Función Maps de RxJS

- Función de transformación que aplica una proyección para cada valor del origen. Aplicamos aquí la programación funcional indicando de forma declarativa una función matemática
- En el ejemplo por cada elemento del array aplica una función que lo eleva al cuadrado

```
import { map } from 'rxjs/operators';

const nums = of(1, 2, 3);

const squareValues = map((val: number) => val * val);
const squaredNums = squareValues(nums);

squaredNums.subscribe(x => console.log(x));

// Logs
// 1
// 4
// 9
```

Función Filter de RxJS

- Emite solo aquellos valores de la secuencia de datos que pasen una condición booleana

```
import { filter, map } from 'rxjs/operators';

const nums = of(1, 2, 3, 4, 5);

// Create a function that accepts an Observable.
const squareOddVals = pipe(
  filter((n: number) => n % 2 !== 0),
  map(n => n * n)
);

// Create an Observable that will run the filter and map functions
const squareOdd = squareOddVals(nums);

// Subscribe to run the combined functions
squareOdd.subscribe(x => console.log(x));
```

RxJS Pipes

- Se pueden usar pipes para enlazar varios operadores
- Permiten así combinar múltiples operadores en una sola función
- La función pipe() toma como argumento las funciones que quieres combinar, y devuelve una nueva función que cuando se ejecuta, va lanzando dichas operaciones de forma secuencial
- El conjunto de operadores pueden considerar como una receta, es decir, la receta no hace nada, tu necesitas invocar a subscribe() para producir el resultado de dicha receta

Ejemplo del Pipe de RxJS

- Filter solo emite solo aquellos valores de la secuencia de datos que pasen una condición booleana. En el ejemplo, solo deja pasar valores impares
- Una vez filtrados, maps aplica la funcion n2 y componen la funicón squareOddVals

```
import { filter, map } from 'rxjs/operators';

const nums = of(1, 2, 3, 4, 5);

// Create a function that accepts an Observable.
const squareOddVals = pipe(
  filter((n: number) => n % 2 !== 0),
  map(n => n * n)
);

// Create an Observable that will run the filter and map functions
const squareOdd = squareOddVals(nums);

// Subscribe to run the combined functions
squareOdd.subscribe(x => console.log(x));
```

Observables en Angular

- Angular 7 hace uso de observables como un interface para manejar un conjunto de operaciones asíncronas. Los casos más importante son:
  - The EventEmitter class de las propiedades de salida de los componentes extienden de Observable
  - El HTTP Module usa los observables para manejar las peticiones AJAX y sus respuestas
  - El Router Module y los Reactive Forms usan observables para escuchar y responder a los eventos de entrada del usuario

HTTP Client

- La mayoría de las aplicaciones Front-end se comunican con servicios backend mediante el protocolo HTTP
- Los navegadores modernos suportan 2 APIs para hacer peticiones HTTP: XMLHttpRequest y el Fetch() API
- El HTTPClient de @ionic/angular ofrece un cliente simplificado HTTP API para aplicaciones basado en XMLHttpRequest
- Incorpora beneficios que influyen características de testeabilidad, peticiones y respuestas tipadas, y intercepción de respuestas
- La ventaja principal es el uso de un API basado en objetos Observables que simplifica el tratamiento del flujo de información que viene del servidor

Uso de HTTPClient

- Debemos importar el módulo HTTPClientModule para hacer uso del mismo en nuestro provider
- Lo podemos hacer en el módulo raiz AppModule, o solo en aquellos donde lo vamos a utilizar

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
```

Uso de HTTPClient

- Una vez importado el HTTPClientModule, se debe inyectar el HTTPClient en el servicio donde lo vayamos a utilizar

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

Invocación a un servicio (Get)

- Normalmente vamos a hacer uso de las peticiones get tipadas. Esto requiere definir un interfaz para obtener el JSON devuelto de forma más sencilla

```
export interface CardDeck{
    name: string;
    types:string[];
}
```

- Definimos la URL a la cual vamos acceder mediante una constante de la clase. Incluso en ocasiones si nos piden un token igualmente lo definimos

```
private readonly HS_API_URL = 'https://omgvamp-hearthstone-v1.p.mashape.com';
private readonly API_KEY = 'MeEShCNExemshD1LXXmdNZrosGPYp1HmjyvjsnkDgqCVZD1Ryo';
```

Petición Get

- En nuestra clase de servicio definimos un método por cada tipo de petición
- El retorno de la operación Get será un array tipado Observable

```
public getAllCardDecks(): Observable<CardDeck[]>{

  return this.http.get<CardDeck[]>(this.HS_API_URL+'/info', {headers: this.headers});

}
```

- Para pasar el token utilizamos un objeto tipo HTTPHeaders para enviarlo en la cabecera del request

```
this.headers = new HttpHeaders ({'X-Mashape-Key': this.API_KEY});
```

Ejercicio 2º de HearthStoneApp:

1. Vamos a recuperar del Hearthstone API la información de los diferentes mazos de cartas. Para ello accedemos Web donde nos describen su API REST. https://hearthstoneapi.com/ . ConsumeAPI. Debemos registrarnos en RapidAPI y accedemos a todos los métodos del API
2. Creamos una interfaz ***CardDeck*** (ver pag. 38) en el fichero *card.model.ts* en la carpeta *app/card/shared* para usar un Get tipado
3. Creamos un clase servicio llamada *card.service.ts* utilizando para ello el *ionic generate*. Elegimos la opción service, y le llamamos "***card***"
4. Importamos las clases Observable de la librería 'rxjs', las HttpClient y HttpHeaders de '@angular/common/http' y la clase ***CardDeck*** del fichero './card.model'
5. Definimos 2 constantes para guardar la dirección URL del API -> HS_API_URL y el token de seguridad -> API_KEY (ver pag. 38)
6. Se define un objeto ***HttpHeaders***, que pase en el campo '**X-RapidAPI-Key**' la constante API_KEY (ver pag. 39)
7. En el constructor del servicios inyectamos un ***HTTPClient*** con el nombre http (Ver pag. 37)
8. Definimos un método ***getAllCardDecks*** que invoque método 'info' del HearthStone API. Devolviendo un array Observable de tipo ***CardDeck*** (ver pag. 39)

Ejercicio 2º de HearthStoneApp: (Cont.)

9. Pasamos como parámetro del constructor de la página ***CardDecksPage*** al servicio ***CardService***
10. En el modulo de la página incorporamos en providers a ***CardService*** y en los imports el módulo ***HttpClientModule*** (ver. Pag 19)
11. Eliminamos el método onInit de la página CardDecksPage
12. Definimos un array **ALLOWED_DECKS** con los mázos válidos: `['classes', 'factions', 'qualities', 'types', 'races'];`
13. Redefinimos el array CardDecks para que sea de tipo CardDeck y vacío:
    ```
    public cardDecks:CardDeck [] = [];
    ```
14. Definimos un método ***extractAllowedDecks*** que nos filtre el array recibido con los mazos válidos (ver pag. 42)
15. Definimos un método ***getCardDeck*** para que se suscriba al método ***getAllCardDecks*** del servicio CardService (ver. Pag 42)
16. Definimos un componente ***card-list*** en la carpeta app/components con el ***ionic generate***
17. En el componente definimos dos input properties denominados *items* es un array de tipo *any* y ***listName*** de tipo ***string***
18. Definimos la vista del componente card-list (ver pag. 43) enlazando con dichas propiedades
19. *Añadimos en la sección declarations del módulo de la página, la referencia al componente* ***CardListComponent*** *(ver pag. 19)*
20. Añadimos la etiqueta del componente ***app-card-list*** desde ***card-deck-page*** que recorra la lista de cardDecks con un ***ngFor*** *(ver pag. 4)*

Conversión de datos desde API

- Necesitamos convertir los datos que nos envían desde el servicio info del API
- Solamente queremos mostrar un subconjunto de las listas que recibimos del info
- Usamos la función Push para guardar crear un nuevo array con solo los que nos interesan un conjunto de mazos en un array ALLOWED_DECKS

```
extractAllowedDecks (cardDecks: CardDeck[]){
  this.ALLOWED_DECKS.forEach ((deckName:string) => {
    this.cardDecks.push({name:deckName, types:cardDecks[deckName]})
  })
```

- La subscripción al servicio nos quedaría así:

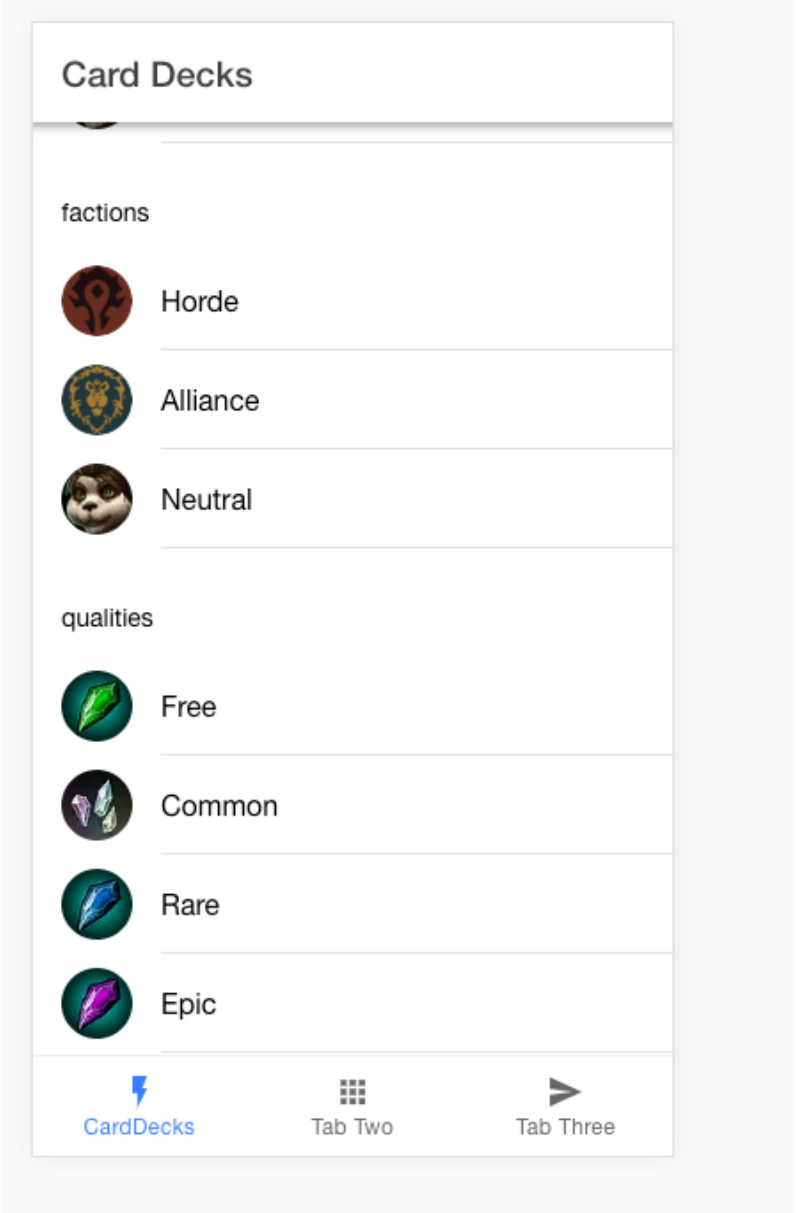
```
private getCardDecks() {
  this.cardService.getAllCardDecks().subscribe(
    (cardDecksService:CardDeck[])=> {
      this.extractAllowedDecks(cardDecksService);
    }
  )
}
```

Componente card-list

- La vista del componente se enlaza con sus dos propiedades

```
<ion-list *ngIf="items.length > 0">
  <ion-list-header>
     <ion-label>{{listName}}</ion-label>
  </ion-list-header>
  <ion-item *ngFor="let item of items">
    <ion-avatar slot="start">
      <img [src]="'assets/image/' + item + '.png'">
    </ion-avatar>
    <ion-label>
      <h2>{{item}}</h2>
    </ion-label>
  </ion-item>
</ion-list>
```

Solución final



Documentación

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