7- Watch : Vigilancia y seguridad en Angular

La **vigilancia** de los datos y la información en tiempo real al usuario son dos pilares del desarrollo con Angular en el lado del navegador. La **seguridad** de los datos realmente es una responsabilidad compartida entre el servidor y el cliente.

Veremos ambos aspectos del desarrollo, pues están muy relacionados con la programación asíncrona y el dominio de los observables. Sentaremos las bases para unas **comunicaciones seguras y fluidas en Angular**.

Partiendo de la aplicación tal cómo quedó en Comunicaciones http en Angular. Al finalizar tendrás una aplicación que comunica a los usuarios cualquier información relevante y que gestiona de forma centralizada las respuestas de un servicio REST.

> Código asociado a este artículo en *GitHub*: AcademiaBinaria/angular-board/7-watch
>
> > Tienes una versión desplegada operativa para probar Angular Board

1. Observables para monitorizar datos

Hemos visto varias técnicas para comunicar información dentro de una aplicación Angular. Empezamos por conocer el flujo entre componentes de una misma rama del DOM. También enviamos datos en los parámetros de una ruta. Y obviamente podemos usar un servicio común para guardar información compartida. Pero en este caso, ¿Cuándo se actualiza? ¿Cómo saber si ha cambiado?. Lo resolveremos con `Observables`.

Para ilustrar este tema vamos a crear un sencillo sistema de notificaciones que informe al usuario. Empezaremos creando un módulo para los propósitos de este laboratorio.

```
ng g m notifications --routing true
ng g c notifications/sender
ng g c notifications/receiver
```

Lo apuntamos al enrutador global `app-routing.module.ts` y asignamos las rutas locales

```
{
  path: 'notifications',
  loadChildren: './notifications/notifications.module#NotificationsModule'
},
```

```
const routes: Routes = [
  {
    path: 'sender',
    component: SenderComponent
  },
  {
```

```
    path: 'receiver',
    component: ReceiverComponent
  },
  {
    path: '**',
    redirectTo: 'sender'
  }
];
```

Por último un enlace en el menú principal `header.component.html` y ya estamos listos para enviar y recibir desde dos componente desacoplados.

```
<a routerLink="notifications" class="button">Notifications</a>
```

Pero antes un poco más de observables.

1.1 Productores de observables

La librería RxJS es enorme y Angular hace un uso extenso de ella. En este tutorial se ha visto desde el punto de vista del consumidor. Es decir, nos hemos suscrito a fuentes observables. Para avanzar tendremos que poder emitir, o mejor dicho producir, información.

Of y from

Los constructores más sencillos de la librería son **funciones que emiten valores estáticos** o secuencias a intervalos regulares. Para familiarizarte con ellos te propongo que juegues con código como este:

```
value$ = of(new Date().getMilliseconds());
value$.subscribe(r=> console.log(r));
stream$ = from([1, 'two', '***']);
stream$.subscribe(r=> console.log(r));
list$ = of(['N', 'S', 'E', 'W']);
list$.subscribe(r=> console.log(r));
```

Subject y BehaviorSubject

Los anteriores constructores se basan en datos estáticos. Resuelvan algunas situaciones, pero necesitamos algo que emita **cambios dinámicos**. Y eso se realiza con los *Subjects*, una especie de emisores temáticos a los que suscribirse.

Hay varios tipos pero para empezar nos vamos a fijar en dos: el `Subject()` y el `BehaviorSubject(initialData)`. La diferencia es que el primero sólo emite las cosas según ocurren. Si alguien se suscribe tarde no conocerá el pasado. Esto suele generar problemas de sincronización. El *Behavior* en cambio notifica el último valor conocido a cualquiera que se suscriba. De esa forma no importa si te suscribes antes o después de la obtención de un dato.

Juega con el siguiente ejemplo:

```
const data = {name:'', value:0};

const need_sync$ = new Subject<any>();
// on time
need_sync.subscribe(r=> console.log(r));
need_sync.next(data);
// too late
need_sync.subscribe(r=> console.log(r));

const no_hurry$ = new BehaviorSubject<any>(this.data);
// its ok
no_hurry.subscribe(r=> console.log(r));
no_hurry.next(data);
// its also ok
no_hurry.subscribe(r=> console.log(r));
```

1.2 Un Store de notificaciones

Usaremos el `BehaviorSubject` como notificador principal entre componentes de Angular. Por ejemplo podemos montar un sistema de notificaciones sencillo. Empecemos por crear un servicio:

```
ng g s notifications/notificationsStore
```

> Para adaptarnos a la nomenclatura usada por patrones de gestión de estado más avanzados como es **Redux**, usaré el siguiente convenio: *Store* como almacén, *select$()* como publicador de cambios observable y *dispatch* como encargado de procesar una acción de cambio de estado.

```
export class NotificationsStoreService {
  private notifications = [];

  private notifications$ = new BehaviorSubject<any[]>([]);

  constructor() {}

  public select$ = () => this.notifications$.asObservable();

  public dispatch(notification) {
    this.notifications.push(notification);
    this.notifications$.next([...this.notifications]);
  }
}
```

Este servicio es la implementación más sencilla posible de un gestor de estados. Cabe destacar que :

- Mantienen el estado privado para evitar manipulaciones
- Recibe de forma controlada las acciones de cambio
- Emite clones del estado
- Expone observables para que se suscriban los interesados.

1.3 Desacoplados pero conectados

Una vez que hemos centralizado el control de cambios de una parte de la aplicación, es hora de que lo usen los componentes o servicios involucrados. Solo necesitan recibir la instancia vía dependencia. **No hay más acoplamiento entre emisores y receptores.**

Emisión

Veamos un ejemplo, un tanto forzado, consistente en dos componentes que se comunican sin conocerse. Esta es la vista con un formulario para enviar mensajes

```
<h2>
  Notes sender
</h2>
<form>
  <fieldset>
    <section>
      <label for="note">Note</label>
      <input name="note"
             [(ngModel)]="note" />
    </section>
  </fieldset>
  <button (click)="send()">Send</button>
</form>
<a [routerLink]="['../receiver']">Go to receiver</a>
```

La parte interesante está en el controlador. Dependencia y uso del servicio del almacén de notificaciones

```
export class SenderComponent implements OnInit {
  public note = '';

  constructor(private notificationsStore: NotificationsStoreService) {}

  ngOnInit() {}

  public send() {
    this.notificationsStore.dispatch(this.note);
  }
}
```

Recepción

La recepción es igual de sencilla. En la vista pondremos un listado de notificaciones que se alimenta de un array emitido por un observable.

```
<h2>
  Notes receiver
</h2>
<ul>
  <li *ngFor="let note of notes$ | async">{{ note | json }}</li>
</ul>
<a [routerLink]="['../sender']">Go to sender</a>
```

Y en el controlador reclamamos la misma dependencia para el uso del servicio del almacén de notificaciones.

```
export class ReceiverComponent implements OnInit {
  public notes$;

  constructor(private notificationsStore: NotificationsStoreService) {}

  ngOnInit() {
    this.notes$ = this.notificationsStore.select$();
  }
}
```

Es importante recalcar que **no importa el orden de suscripción**. Estos dos componentes podrían *vivir* en módulos distintos, verse en la misma página o inicializarse en cualquier orden... El receptor se entera siempre de todos los cambios; y además recibe el último estado conocido nada más suscribirse.

2. Interceptores para gestionar errores

Hemos conocido a los interceptores y vemos su potencial para manipular las respuestas de una API. Quizá uno de los usos más frecuentes se el de **centralizar la gestión de errores**. Veamos como hacerlo usando el conocimiento de los observables.

Para empezar hay que generar un servicio...

```
ng g s notifications/errorInterceptor
```

luego hay que hacerle cumplir la interfaz `HttpInterceptor`...

```
export class ErrorInterceptorService implements HttpInterceptor {
  constructor() {}

  public intercept(req: HttpRequest<any>, next: HttpHandler)
```

```
    : Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

y para finalizar lo proveemos hacia el `HttpClient` invirtiendo el control.

```
@NgModule({
  declarations: [SenderComponent, ReceiverComponent],
  imports: [
    CommonModule,
    NotificationsRoutingModule,
    HttpClientModule,
    FormsModule
  ],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: ErrorInterceptorService,
      multi: true
    }
  ]
})
export class NotificationsModule {}
```

2.1 El operador catchError

Volvemos a los observables y los operadores canalizables en `.pipe()`. Durante su ejecución un *stream* de observables puede emitir valores correctos, una señal de finalización... y cómo no, errores. El método `.subscribe(ok, err, end)` y operadores como `.map(ok, err, end)` admiten hasta tres *callbacks* que se llamarán según los tipos de sucesos descritos. Pero para tratar el caso concreto de los errores vamos a ver el operador `catchError()`.

Por ejemplo durante la intercepción de respuestas podemos realizar una función específica al recibir un código de error. Dadas estas tres alternativas, escogeremos según la intención o la tecnología que más nos guste.

```
public intercept(req, next) {
   // implementación con .tap()
   return next.handle(req).pipe(tap(null, err=>console.log(err)));
   // implementación con catchError retornando nulo
   return next.handle(req).pipe(catchError(err => of(null)));
   // implementación con catchError re-lanzando el error
   return next.handle(req).pipe(catchError(err => throwError(err)));
}
```

2.2 Gestión centralizada de errores

Quizás una de las más usadas sea auditar el error y reenviarlo al llamante original por si quiere hacer algo más con el mismo.

```
public intercept(req, next) {
  return next.handle(req).pipe(catchError(this.handleError));
}

private handleError(err) {
  const unauthorized_code = 401;
  let userMessage = 'Fatal error';
  if (err instanceof HttpErrorResponse) {
    if (err.status === unauthorized_code) {
      userMessage = 'Authorization needed';
    } else {
      userMessage = 'Comunications error';
    }
  }
  console.log(userMessage);
  return throwError(err);
}
```

Pero aún mejor que solo escribir en el *log*, sería avisar al usuario; ¿pero dónde y cómo?

3. Un notificador de problemas

La idea es usar el `NotificationsStoreService` desde el interceptor para... en fin, notificar que ha habido un error.

3.1 Emisión mediante el Store

```
// dependencia en el constructor
constructor(private notificationsStore: NotificationsStoreService) {}

public intercept(req, next) {
  // Ojo al bind(this), necesario para no perder el contexto
  return next.handle(req).pipe(catchError(this.handleError.bind(this)));
}

private handleError(err) {
  let userMessage = 'Fatal error';
  // emisión de la notificación
  this.notificationsStore.dispatch(userMessage);
}
```

3.2 Recepción desacoplada del interceptor

Y ahora ya sólo queda suscribirse a los eventos y mostrarlos al usuario. Por ejemplo, desde el `ReceiverComponent`, podemos lanzar llamadas que sabemos que darán problemas y esperar pacientemente el fallo para mostrarlo al usuario.

```
<button (click)="forceError()">Force http Error</button>
```

```
public forceError() {
  const privateUrl = 'https://api-base.herokuapp.com/api/priv/secrets';
  this.httpClient.get(privateUrl).subscribe();
  const notFoundUrl = 'https://api-base.herokuapp.com/api/pub/items/9';
  this.httpClient.get(notFoundUrl).subscribe();
}
```

Tenemos ahora a nuestro usuario puntualmente informado de todo lo que sucede. Hemos utilizado **patrones de arquitectura de software** como el observable y la inversión del control. El resultado es una serie de componentes y servicios poco acoplados que intercambian información en tiempo real.

Pero hay temas más avanzados con los que continuar. Sigue esta serie para introducir más temas de seguridad y crear tus formularios reactivos con Angular mientras aprendes a programar con Angular.

> Aprender, programar, disfrutar, repetir. -- *Saludos, Alberto Basalo*