Class 18 - Understanding Observables && Course Project

Class 18 Course Content

Lesson Outline

Today we will learn:

1. What Asynchronous code is.
2. What an Observable is.
3. How to subscribe to Observables.
4. What RxJS Operators are.
5. How to map over code.
6. The basics of Pipes.
7. What a Subject is in Angular.

Lesson Notes

- **Asynchronous:** *Asynchronous* code is any piece of logic that does not execute right away because it needs to wait for something else to happen first. You can understand asynchronous code better by knowing how Javascript works under the hood. Javascript is a "Single-Threaded-Language," meaning that it can only execute one piece of code at a time. So when it runs into, let us say, an HTTP Request... it will be held up for 5-10 seconds! Waiting this long to respond would be a horrible user experience, so what happens is that the Javascript Evaluator says: "This might take a bit. I will put a note over here and insert myself when I finish this complex task". Javascript knows to do this when we write *Asynchronous* code.
- **Observable:** An *Observable* is an interface used to handle various everyday asynchronous or event operations. They provide support for passing messages between different sections of your application. Observables are a stream of data that you "subscribe" to receive info when that data changes.
- **Observer:** An *Observer* is a way to "subscribe" to Observable events. Depending on where the Observable is placed, you can write code to handle the data, handle the error, and handle the completion.
- **Subjects:** *Subjects* are unique *Observables* that act as both the "observer" and the "observable." Subjects allow us to emit new values to the subscription stream using the `next()` method.

Course Project Steps

STEP 1: Replacing EventEmitter with Subjects

bookshelf/bookshelf.service.ts file:

- Import `{ Subject }` from "rxjs".
- Remove the "EventEmitter".
- Add a new `Subject<Book>()`.
- When we save a book, we should use the "next" method to replace "emit". (Change the "emit" to "next" in the *shared/book* component as well.)

```
bookSelected = new Subject<Book>()
bookListChanges = new Subject<Book[]>()

// . . .

saveBook(book: Book) {
    this.myBooks.push(book);
    this.bookListChanged.next(this.myBooks.slice());
}

removeBook(idx: number) {
    if (idx !== -1) {
      // We have a book at that index
      this.myBooks.splice(idx, 1);
      this.bookListChanged.next(this.myBooks.slice());
    }
}
```

bookshelf/book-details/book-details.component.ts:

- Remove the `onSelected()` method because we no longer need it.

bookshelf/book-list/book-list.component.ts:

- Remove the `handleBookSelected()` method.
- *Note*: When we emit an event, we cannot see those changes within a separate component even if they are subscribed to it. So, let us create a notification component to demonstrate the subscription concept.

STEP 2: Adding an Alert Notification Component

Terminal:

- Create a new component in the shared folder called "notification".

```
ng g c shared/notification
```

library/library.component.html file:

- Add your `<app-notification>` tag to the bottom of the page.

shared/notification/notification.component.ts:

- Inject the "bookshelfService" in the constructor.
- Subscribe to the "bookshelfService.bookSelected" method inside "ngOnInit()", and alert to the browser of the title & author.

```
constructor(private bookshelfService: BookshelfService) {}

ngOnInit(): void {
this.bookshelfService.bookSelected.subscribe(data=>{
    console.log(data);
    alert(`title: ${data.title}\n author: ${data.author}`)
});
}
```

- Refactor your code to create a subscription on initialization by storing the "Subscription" in a separate variable.
- Unsubscribe to the changes in "ngOnDestroy()" to avoid creating multiple instances of the "notification" component.

```
export class NotificationComponent implements OnInit, OnDestroy {
  private bookChangeSub: Subscription;

  // . . .

  ngOnInit(): void {
    this.bookChangeSub = this.bsService.bookSelected.subscribe((data) => {
      // . . .
    });
  }

  // . . .

  ngOnDestroy() {
    this.bookChangeSub.unsubscribe();
  }
}
```

STEP 3: Cleaning Up Our Code

bookshelf/bookshelf.component.ts file:

- Remove all the logic inside this component.

```
export class BookshelfComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {}
}
```

Additional Notes

Class Exercise - Observables

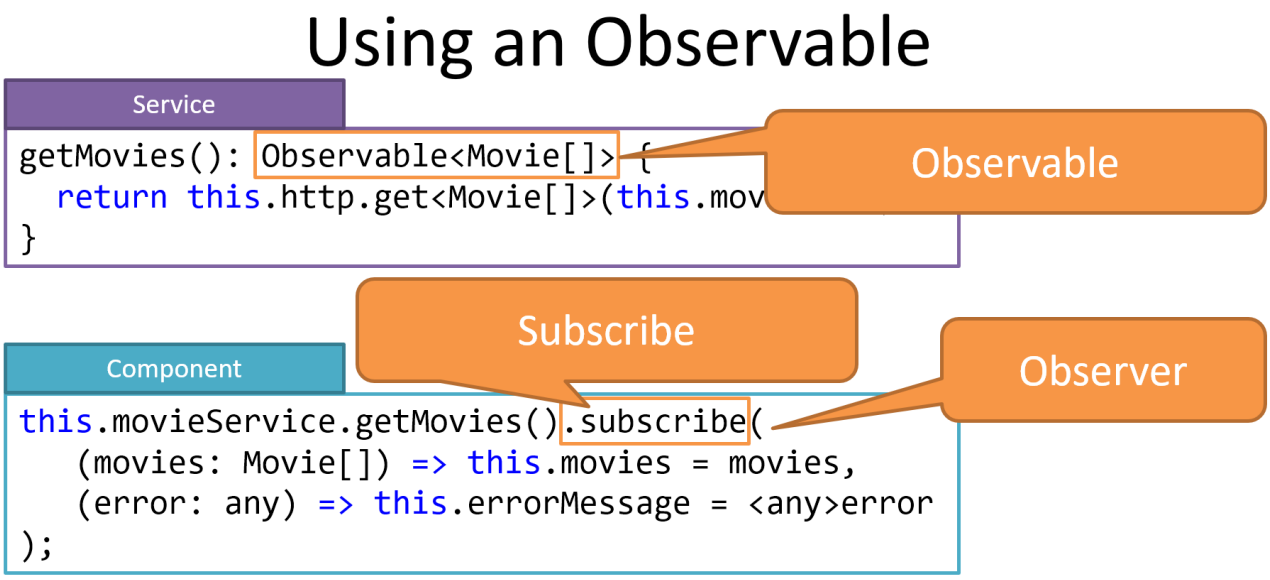
Github Repo Starting Code

1. Create a new Subject in the PillowCaseService that emits when the myCandies array updates.
2. Subscribe to that change and update the local myCandies array.
3. Create a variable of type "Subscription", store the observable under inside, and use ngOnDestroy to unsubscribe when the component is destroyed.
4. Create a new Subject in the CandyStoreService that emits when the candiesForSale array updates.
5. Subscribe to that change and update the local candiesForSale array.
6. Create a variable of type "Subscription", store the observable under inside, and use ngOnDestroy to unsubscribe when the component is destroyed.

Observable Terminology Expanded

What is an Observable?

- Observable is a stream of events or data. In Angular, an Observable is just an object we import from "rxjs".
- An Observable can emit data programmatically or by a button, for example.
- Observables can return values asynchronously but also synchronously.
- An observable is like a placeholder; by subscribing to it, you are saying, "when this happens, do this" It can happen anytime asynchronously.
- this.route.params is an example of an observable(stream of data)



What is an Observer?

- An Observer is basically something that subscribes to Observable. With observers, we can handle data, handle errors and handle completion.

What is Subscribing?

- Subscribing "kicks off" the observable stream. The stream will not emit values without a subscribe method (or an async pipe). It is similar to subscribing to a newspaper or magazine, and you will not start getting them until you subscribe.
- The subscribe method takes an observer.

What does the Unsubscribe method do?

- The unsubscribe method cancels observable executions. Doing this is good when you do not want many instances happening at once to preserve resources. Some observables have a built-in angular configuration unsubscription that they will handle independently.
- You can build your observables with the Observable object imported from RxJs.

What are RxJs Operators?

- RxJs operators are simply methods that you can use on Observables (and Subjects) that allow you to change the original observable in some manner and return a new observable. Examples of rxjs: map, filter, concat, ect

What does Map do?

- Map is used to transform data before the observable transmits the data to what is being subscribed to.
- *Note*: You can filter out what to transmit with the filter operator

What does Pipe do?

- Pipe is an observable method used to combine RxJS operators to compose asynchronous operations

What is a Subject?

- Subject is a special type of Observable in RxJs Library in which we can send our data to other components or services.
- Although subjects can be more efficient than event emitters, subjects are found to be used as cross-component event emitters.
- Subjects are by default "asynchronous" and supported by Angular for Observables.

Resources

- Angular Docs - Observables in Angular
- Angular Docs - Using Observables to Pass Values