Angular Services

What are Services ?

Dependency Injection

RxJS Basics

What are Services ?

The user interface of an Angular application is developed by embedding several components into the main component.

However, these components are generally used only for rendering purposes. In other words, They are only used to define what appears on the user interface.

Other tasks, like data and image fetching, network connections, database management, are not performed through components.

Then how are these tasks achieved ? And what if more than one component performs similar tasks ? Well, Services take care of this.

What are Services ?

As the name suggests, **Angular Services** are used to offer “services” to various components. They allow you to write reusable and organized code that can be accessed by multiple components **without duplicating it**.

They help us implement the **MVC** pattern such that:

Components act as the View, managing the user interface and interacting with the user.

Services act as the Controller, containing the business logic and connecting the View (Components) to the data.

Model is the application's data,represented as objects.

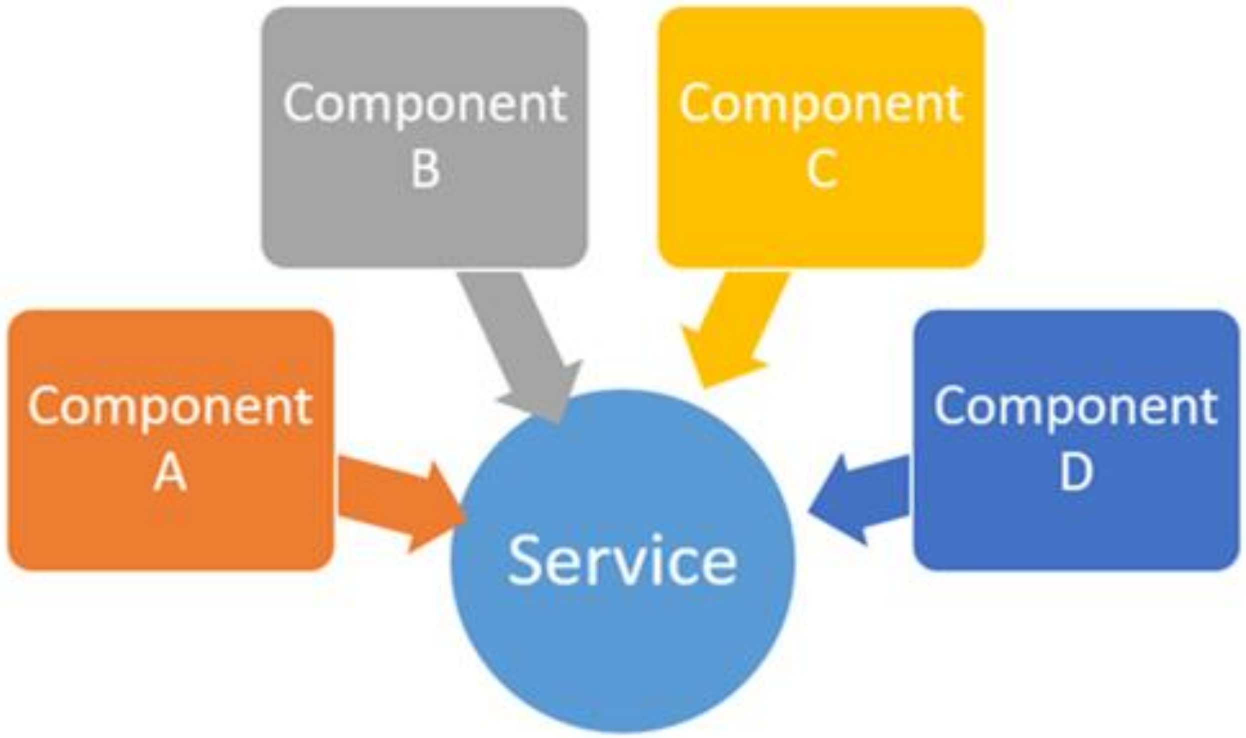
When to use Services ?

Shared Data: When multiple components need to access or update the same data, services help centralize and manage it.

Business Logic: To separate business logic (like calculations or decision-making) from your components, keeping components simple and focused.

API Calls: To handle HTTP requests and fetch or send data to a backend.

Reusability: When a specific functionality (like logging or authentication) is used across multiple parts of the app.


Component B
Component C
Component A
Component D
Service

Example Without Services

The **AddNote** component needs access to the note list data from the **NoteList** component. To share this data, we'll use a **service** that both components can access.

```
export class NoteListComponent {
  notes = ['Note 1', 'Note 2',
'Note 3'];
}
```

```
export class AddNoteComponent {
  note = '';
  addNote() {
    console.log(this.note);
    this.note = '';
  }
}
```

Creating a Service

To create a new service, Use the `ng generate service Notes` command.

A service is a class annotated with **@Injectable**, Which means Angular can **inject** (provide) it into other components or services.

`{ providedIn: 'root' }` registers the service with the **root injector** (the class responsible for creating the service) which makes the service **globally** usable everywhere.

This also makes the service a **Singleton**. Angular creates only **one instance** of the service for the entire app, no matter how many times it’s used.

```
@Injectable({ providedIn: 'root' })
export class NotesService {}
```

Creating a Service

To make a service available only in a **specific module**, remove the `{ providedIn: 'root' }` from its configuration. Then, add the service to the **providers array** of the desired module.

If you add the service to the providers array of the **AppModule (the root module)**, the service will also be **globally** available throughout the entire application.

```
@NgModule({
  declarations: [AppComponent, NoteListComponent, AddNoteComponent],
  imports: [BrowserModule, AppRoutingModule, FormsModule],
  providers: [NotesService], // The services available to this module.
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Notes Service

The NotesService is responsible for managing the note list data both components need.

```
@Injectable({ providedIn: 'root' })
export class NotesService {
  private notes = ['Note 1', 'Note 2', 'Note 3'];
  addNote(note: string) {
    this.notes.push(note);
  }
  getNotes() {
    return this.notes;
  }
}
```

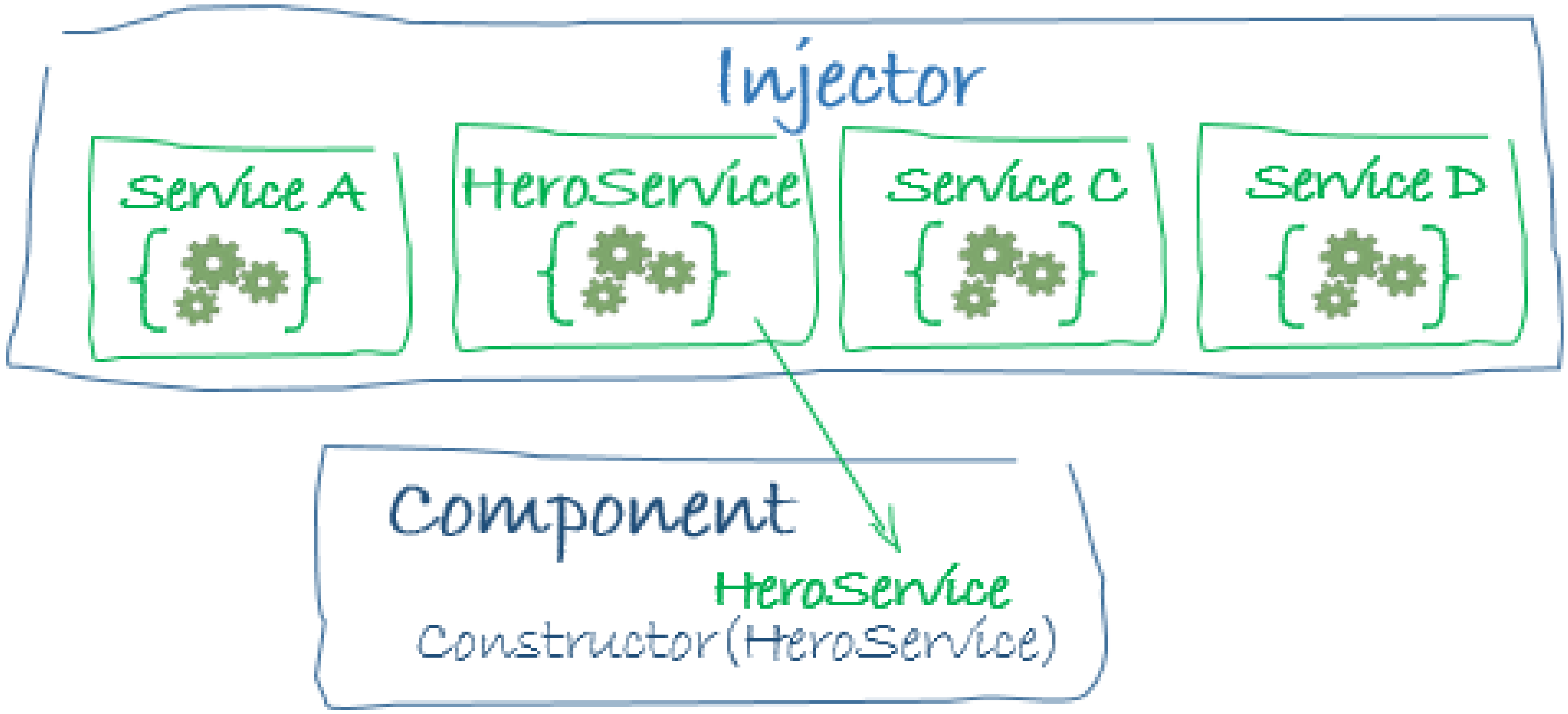
Using a Service

To use a service in a component, Add it as a parameter to the component's constructor. Angular will create an instance of the service and **"inject"** it into the component. This is called **Dependency Injection**.

If you write an access modifier (like private or public) before the parameter in the constructor, Angular makes it a **member variable** of the class. This allows you to use the service anywhere in the component, not just inside the constructor.

```
export class NoteListComponent {
  notes: string[] = [];
  constructor(private notesService:
NotesService){
    this.notes = notesService.getNotes();
  }
}
```

```
export class AddNoteComponent {
note = '';
constructor(private notesService: NotesService)
{}
  addNote() {
    this.notesService.addNote(this.note);
    this.note = '';
  }
}
```


Injector
Service A
HeroService
Service C
Service D
Component
HeroService
Constructor(HeroService)

Notifying the Service Users

If the data inside the NotesServices **changes** we need a way, to notify its users about the new data

```
@Injectable({ providedIn: 'root' })
export class NotesService {
  private notes = ['Note 1', 'Note 2', 'Note 3'];
  addNote(note: string) { this.notes.push(note); }
  getNotes() { return this.notes; }
  filterNotes() { // Replace the notes array with a filtered version.
    this.notes = this.notes.filter((note) => note.includes('Note 1'));
  }
}
```

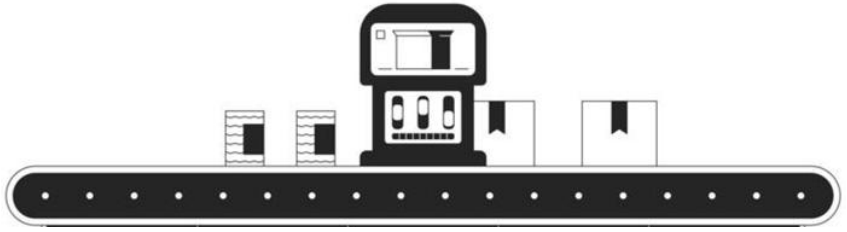
RxJS Basics

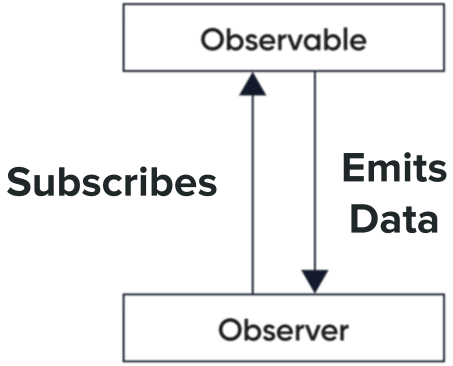
Imagine you have a **factory that produces new items (data)** every once in a while. **RxJS** helps you manage this production line efficiently.

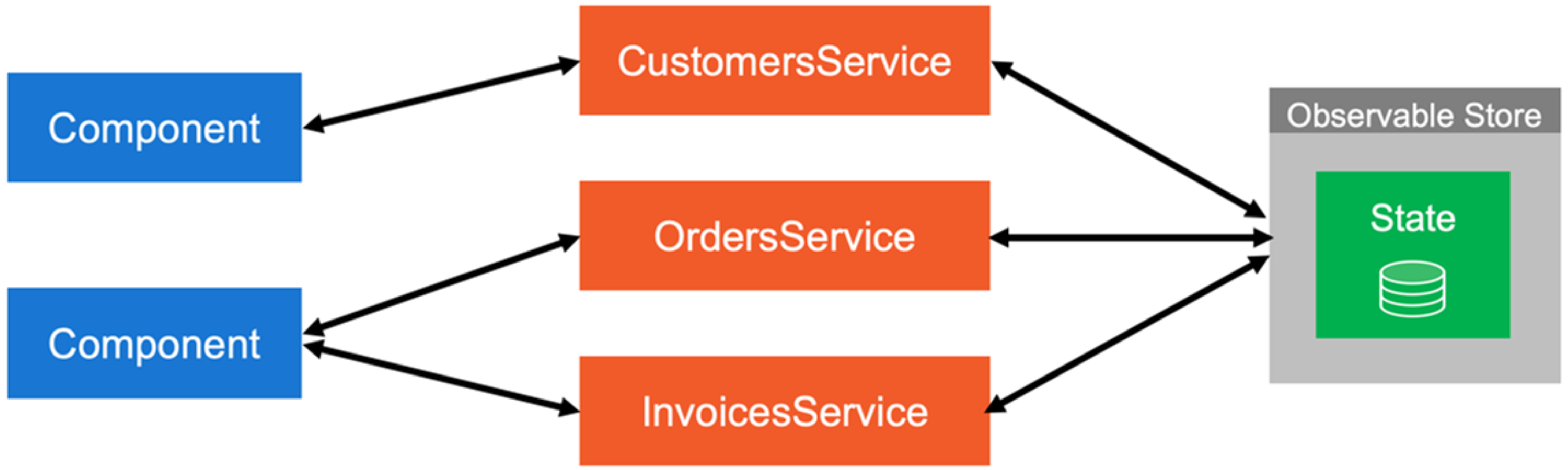
An **Observable** is like a factory production line that **creates items (data) over time**.

It doesn't produce everything at once; it produces items at its own pace.

You can **subscribe** to the production line to start receiving items.






Component
Component
CustomersService
OrdersService
InvoicesService
Observable Store
State

RxJS Observer

An **Observer** is like a worker at the end of the production line.

The Observer **subscribes an observable** to a receives items from the line and decides what to do with them (process, store, or discard them).

Multiple workers can work on the same production line

Creating Observables

The Observable constructor specifies the **type of data** that the observers will receive, it takes a function that will be called when the Observable **is subscribed to.**

The function takes an **observer** object as an argument, which has a next method that can be called to emit a new value to the observer subscribed to the observable.

```
observableNotes = new Observable<string[]>((observer) => {
  observer.next(['New Note 1', 'New Note 2', 'New Note 3']);
  observer.complete();
  return () => console.log("Unsubscribed");
});
```

Creating Observables

If the Observable has finished sending values, it should call the observer's **complete method**.

The function returns an **unsubscribe** method which is called when the Observer unsubscribes from the Observable, It is used to clean up any resources that the Observable is using, it is also called when the **Observable completes**.

```
observableNotes = new Observable<string[]>((observer) => {
  observer.next(['New Note 1', 'New Note 2', 'New Note 3']);
  observer.complete();
  return () => console.log("Unsubscribed");
});
```

Subscribing to an Observable

Observables are lazy, meaning they do not start emitting data **until they are subscribed to**. Subscribing to an observable triggers its execution **synchronously**.

```
const observable = new Observable((observer) => {
    console.log("Start");
    observer.next(1); // Emits data synchronously
    observer.next(2);
    // Completes the observable
    observer.complete();
    observer.next(3); // Ignored
    console.log("End");
    // The cleanup function is called when the
observable is unsubscribed from, or completes.
    return () => console.log("Unsubscribed");
});
```

```
console.log("Before");
const subscription =
observable.subscribe((value) =>
    console.log(value)
);
console.log("After");
// This will trigger the cleanup
function if the observable still has
not completed.
subscription.unsubscribe();
```

Output:

```
Before
Start
1
2
End
Unsubscribed
After
```

Asynchronous Observable

The observable can be used to emit data **asynchronously** using any asynchronous operation, such as setInterval or a fetch request.

```
const observable = new Observable((observer) => {
    let count = 1;
    // Emit data every second
    const intervalId = setInterval(() => {
        observer.next(count++);
    }, 1000);
// Stop emitting data when observers unsubscribe.
    return () => clearInterval(intervalId);
});
```

```
// Subscribing to the observable triggers
the asynchronous data emission.
const subscription =
observable.subscribe((value) =>
console.log(value));
// The observable will stop emitting data
when the subscription is unsubscribed
from.
setTimeout(() =>
subscription.unsubscribe(), 5000); //
Unsubscribes after 5 seconds
```

Output:

```
1
2
3
4
5
```

Observable Creation Functions

Observables can be created using built-in creation functions like **of**.

```
// The of creation function creates an observable that emits a sequence of
values synchronously then completes.
const observable = of(1, 2, 3, 4, 5);
// Subscribing to the observable triggers the synchronous data emission.
// Output: 1, 2, 3, 4, 5
observable.subscribe((value) => console.log(value));
```

Operator Functions

Operator functions can be used to **transform, or filter** the data emitted by an observable.

To use operators, the **pipe function** is used to chain multiple operators together.

When the observable emits data, it is passed through each operator in the pipe.

The pipe function **returns a new observable with the transformed data.**

```
const observables = of(1, 2, 3, 4, 5);
// The map operator transforms the data emitted by the observable.
const transformedObservable = observable.pipe(map((value) => value * 2));
// Subscribing to the transformed observable triggers the data emission.
// Output: 2, 4, 6, 8, 10
transformedObservable.subscribe((value) => console.log(value));
```

BehaviorSubject

A BehaviorSubject is an observable that can emit new values **after it has been created**.

```
const subject = new BehaviorSubject([1, 2, 3]);
// BehaviorSubjects emit the last emitted value immediately upon
subscription.
// Output: [1, 2, 3]
subject.subscribe((value) => console.log(value));
// BehaviorSubjects can emit new values and notify all subscribers.
// Output: [4, 5, 6]
subject.next([4, 5, 6]);
```

Modifying the Notes Service

The NotesService can use a **BehaviorSubject** to manage the note list. When the note list changes, the new list can be emitted through the BehaviorSubject. This ensures all observers automatically receive the updated data.

```
@Injectable({ providedIn: 'root' })
export class NotesService {
  notes = new BehaviorSubject<string[]>(['Note 1', 'Note 2', 'Note 3']);
  addNote(note: string) {
    this.notes.value.push(note);
  }
  filterNotes() {
    // Replace the notes array with a filtered version.
    const filtered = this.notes.value.filter((note) => note.includes('Note 1'));
    this.notes.next(filtered); // Notify all subscribers of the change.
  }
}
```

Subscribing to the BehaviorSubject

Inside the **NoteListComponent** constructor we can subscribe to the behaviour subject to always receive the latest data when it changes.

```
export class NoteListComponent {
  notes: string[] = [];
  constructor(private notesService: NotesService) {
    notesService.notes.subscribe((notes) => {
      this.notes = notes;
    });
  }
  filterNotes() {
    this.notesService.filterNotes();
  }
}
```

Unsubscribing From Observables

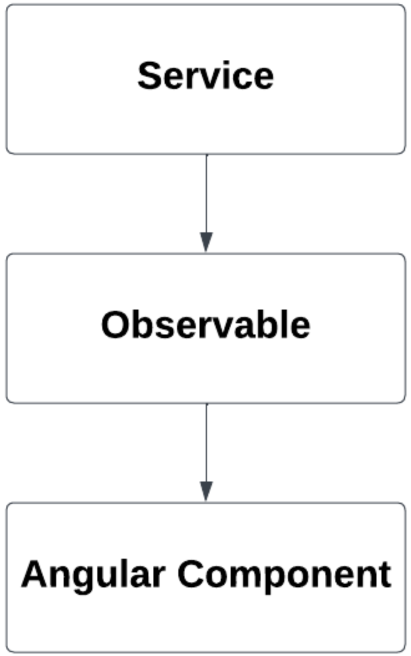
When a component subscribes to an observable in a service, the observable keeps a reference to the component (subscriber).

If the component is destroyed (e.g., navigating to another page), the observable still holds this reference.

Since services are singletons and live throughout the app, they also keep a reference to the observable.

This prevents the component from being garbage collected, **causing memory leaks with observables that never complete**.

To fix this, you must unsubscribe in the **ngOnDestroy** method to break the connection and allow the component to be cleaned up.



Unsubscribing From Observables

```
export class NoteListComponent {
  notes: string[] = [];
  subscription: Subscription;
  constructor(private notesService: NotesService) {
    // Store the subscription so we can unsubscribe later.
    this.subscription = notesService.notes.subscribe((notes) => {
      this.notes = notes;
    });
  }
  ngOnDestroy() {
     this.subscription.unsubscribe(); // Unsubscribe when the component is destroyed.
  }
}
```

Async Pipe

Keeping track of observable subscriptions manually can be tedious. Instead, you can use the **Async Pipe** in the component template. The Async Pipe **automatically handles subscribing to the observable and also unsubscribes when the component is destroyed**, making your code cleaner and prevents memory leaks.

```
export class NoteListComponent {
  notesObservable:
Observable<string[]>;
  constructor(private notesService:
NotesService) {
this.notesObservable =
notesService.notes;
  }
}
```

```
<ul>
  @for (note of
notesObservable | async; track
note) {
  <li>{{ note }}</li>
  }
</ul>
```

Hands On

Create an Angular application with two components and a service:

SendMessageComponent: allows the user to send a string message using a service.

Service: The service holds the message using a **BehaviorSubject**.

The BehaviorSubject keeps the latest message and automatically notifies all subscribers (components) whenever the message changes.

MessageComponent: listens for updates from the BehaviorSubject in the service.

It displays the latest message sent by the SendMessageComponent.

Assignment

Build an Angular app with two main components: one for user registration and one for displaying the registered users.

RegisterComponent:

- Includes a form with input fields for username and password.
- A "Sign Up" button to add new users.
- Shows an error message if the username already exists.

UserListComponent:

- Displays a list of registered usernames.
- Highlights usernames in red if their password is shorter than 5 characters.
- Allows users to be deleted when their username is clicked.

Assignment

To submit the assignment:

- First create an account on StackBlitz
- Then create a **fork** of this starter template
- Implement the assignment in your forked version
- Copy your StackBlitz link using the share button
- Submit the link in this form
- This is an **individual** assignment, and its deadline is 28/11/2025
- **Any submission flagged for cheating will receive a mark of 0**

Thank You ;)

Any Questions ?