ngOnDestroy

`ngOnDestroy` is a lifecycle hook in Angular that is called just before a component or directive is destroyed. It provides an opportunity to perform cleanup tasks such as unsubscribing from observables, clearing timers, or releasing resources. This hook is particularly useful for preventing memory leaks and ensuring that resources associated with a component are properly released.

Here's a detailed explanation of `ngOnDestroy`:

Definition:

- **Name:** `ngOnDestroy`
- **When is it called:** Just before a component or directive is destroyed.
- **Use Case:** Perform cleanup tasks, unsubscribe from observables, and release resources.

How to Implement `ngOnDestroy`:

1. **Import the Necessary Modules:**

```
import { Component, OnDestroy } from '@angular/core';
```

2. **Implement the OnDestroy Interface:**

```
export class YourComponent implements OnDestroy {
  // Component properties and methods

  ngOnDestroy() {
    // Cleanup logic
  }
}
```

Example Usage:

Let's consider an example where a component subscribes to an observable and needs to unsubscribe when the component is destroyed:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Observable, Subscription } from 'rxjs';

@Component({
  selector: 'app-data-display',
  template: '<p>Data: {{ data }}</p>',
})
export class DataDisplayComponent implements OnInit, OnDestroy {
```

```
  data: string;
  dataSubscription: Subscription;

  ngOnInit() {
    // Subscribe to an observable
    this.dataSubscription = this.dataService.getData().subscribe(result => {
      this.data = result;
    });
  }

  ngOnDestroy() {
    // Unsubscribe from the observable to prevent memory leaks
    this.dataSubscription.unsubscribe();
  }
}
```

In this example:

- The `DataDisplayComponent` implements `OnInit` to subscribe to an observable during component initialization.
- The `ngOnDestroy` method is implemented to unsubscribe from the observable when the component is about to be destroyed.

Key Points:

1. **Resource Cleanup:**
    - `ngOnDestroy` is the ideal place to perform cleanup tasks, such as releasing resources, unsubscribing from observables, or canceling ongoing operations.
2. **Avoiding Memory Leaks:**
    - Unsubscribing from observables in `ngOnDestroy` is crucial to prevent memory leaks. Failing to unsubscribe can lead to ongoing subscriptions even after the component is destroyed.
3. **Clearing Timers:**
    - If you have set up timers using functions like `setTimeout` or `setInterval`, `ngOnDestroy` is the place to clear those timers to avoid unintended side effects.
4. **Cleaning Up External Resources:**
    - Release external resources, connections, or subscriptions that the component may have established during its lifecycle.

Example in a Component:

```
import { Component, OnDestroy } from '@angular/core';

@Component({
```

```
  selector: 'app-example',
  template: '<p>Example Component</p>',
})
export class ExampleComponent implements OnDestroy {
  intervalId: any;

  constructor() {
    // Set up an interval timer
    this.intervalId = setInterval(() => {
      console.log('Interval tick');
    }, 1000);
  }

  ngOnDestroy() {
    // Clear the interval timer to prevent memory leaks
    clearInterval(this.intervalId);
  }
}
```

In this component example, the `ngOnDestroy` method is used to clear an interval timer set up in the constructor, preventing the timer from continuing after the component is destroyed.

The `ngOnDestroy` lifecycle hook in Angular is used to perform cleanup tasks and release resources just before a component or directive is destroyed. Here are some common scenarios where you might use `ngOnDestroy`:

1. **Unsubscribing from Observables:**
    - If your component subscribes to observables, it's important to unsubscribe to prevent memory leaks. You can use `ngOnDestroy` to unsubscribe from observables.

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

2. **Clearing Timers:**
    - If your component uses timers created with `setTimeout` or `setInterval`, you should clear those timers in `ngOnDestroy` to avoid ongoing operations after the component is destroyed.

```
ngOnDestroy() {
  clearInterval(this.intervalId);
}
```

3. **Cleaning Up External Resources:**
    - If your component establishes connections to external resources (e.g., WebSocket connections), releases resources, or manages state outside Angular's control, use `ngOnDestroy` to clean up those resources.

```
ngOnDestroy() {
  this.externalResource.disconnect();
}
```

4. **Canceling HTTP Requests:**
    - If your component makes HTTP requests, you might want to cancel or unsubscribe from pending requests when the component is destroyed.

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

5. **Destroying Components with NgRx Effects:**
    - In NgRx applications, components that are connected to the store through NgRx effects might need to clean up resources or unsubscribe when the component is destroyed.

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

6. **Closing Modal Dialogs or Popovers:**
    - If your component is used as a modal dialog or popover, you might want to close or destroy these UI elements when the parent component is destroyed.

```
ngOnDestroy() {
  this.modalService.closeDialog();
}
```

7. **Resetting Global State:**
    - If your component modifies global state (e.g., through a service), use `ngOnDestroy` to reset or clean up that state when the component is destroyed.

```
ngOnDestroy() {
  this.globalStateService.resetState();
}
```

8. **Detaching Event Listeners:**
    - If your component attaches event listeners directly to the DOM or external elements, use `ngOnDestroy` to detach those event listeners.

```
ngOnDestroy() {
  window.removeEventListener('scroll', this.scrollHandler);
}
```

Remember that `ngOnDestroy` is not automatically called by Angular; it's your responsibility to implement it in your component or directive. Using `ngOnDestroy` ensures that you release resources and clean up properly, contributing to better memory management in your Angular application.