Angular

State Management:

Parent Component:

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

@Component({
selector: 'app-parent',
standalone: true,
template: `
<h2>Parent Component</h2>
<app-child [message]="parentMessage" (messageUpdated)="onMessageUpdated($event)"></app-child>
<p>Parent Message: {{ parentMessage }}</p>
`
})
export class ParentComponent {
parentMessage: string = 'Initial Parent Message';

onMessageUpdated(message: string) {
this.parentMessage = message;
}
}
```

Child Component:

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
selector: 'app-child',
standalone: true,
template: `
<h2>Child Component</h2>
<p>Received Message from Parent: {{ message }}</p>
<button (click)="updateMessage()">Update Parent Message</button>
`
})
export class ChildComponent {
@Input() message: string;
@Output() messageUpdated: EventEmitter<string> = new EventEmitter<string>();

updateMessage() {
const newMessage = 'New Message from Child';
this.messageUpdated.emit(newMessage);
}
}
```

Component Lifecycle Hooks:

1. ngOnInit():

This hook is called once after Angular initializes the component's data-bound properties. It's a good place to initialize component properties or fetch initial data.

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

@Component({
selector: 'app-home',
standalone:true,
templateUrl: './home.component.html',
styleUrls: ['./home.component.css'],
})
export class HomeComponent implements OnInit {
ngOnInit(): void {
console.log('OnInit Called');
}
}
```

2. ngOnChanges():

The ngOnChanges hook is called when one or more input properties of a component change. Updating your component's state is made convenient by responding to input changes.

```
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';

@Component({
selector: 'app-home',
standalone:true,
templateUrl: './home.component.html',
styleUrls: ['./home.component.css'],
})
export class HomeComponent implements OnChanges {
@Input() inputMessage: string = '';

ngOnChanges(changes: SimpleChanges): void {
console.log(changes);
}
}
```

3. ngDoCheck():

Angular may not always be able to detect or address changes on its own. In such cases, it is important to identify and respond to these changes.

```
import {
Component,
Input,
OnChanges,
OnInit,
SimpleChanges,
} from '@angular/core';

@Component({
selector: 'app-home',
templateUrl: './home.component.html',
styleUrls: ['./home.component.css'],
})
export class HomeComponent implements OnChanges, OnInit {
changeCount: number = 0;
ngOnInit(): void {
console.log('OnInit Called');
}
@Input() inputMessage: string = '';

ngOnChanges(changes: SimpleChanges): void {
console.log(changes);
}
ngDoCheck(): void {
this.changeCount++;
console.log('counter ' + this.changeCount);
}
}
```

4. ngAfterContentInit ():

Angular may not always be able to detect or address changes on its own. In such cases, it is important to identify and respond to these changes.

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

```
@Component({
selector: 'app-example',
template: '<ng-content></ng-content>',
})
export class ExampleComponent implements AfterContentInit {
ngAfterContentInit(): void {
// Access and initialize content children here.
}
}
```

NgAfterContentChecked

After Angular checks the content that is projected into a directive or component, the `ngAfterContentChecked()` will respond accordingly.

```
import { Component, ContentChildren, QueryList, AfterContentChecked } from '@angular/core';

@Component({
selector: 'app-tab',
template: `
<div class="tab">
<ng-content></ng-content>
</div>
`,
})
export class TabComponent implements AfterContentChecked {
ngAfterContentChecked() {
console.log('Content inside the tab checked or changed.');
}
}

@Component({
selector: 'app-tabs',
template: `
<div class="tabs">
<ng-content></ng-content>
</div>
`,
})
export class TabsComponent {}

@Component({
selector: 'app-root',
template: `
<app-tabs>
<app-tab>
<h2>Tab 1</h2>
<p>Content for Tab 1</p>
</app-tab>
<app-tab>
<h2>Tab 2</h2>
<p>Content for Tab 2</p>
</app-tab>
</app-tabs>
`,
})
export class AppComponent {}
```

```
ngAfterContentChecked() {
 console.log('Content inside the tab checked or changed.');
 }
```

NgAfterViewInit

After the component's views and child views, or the view containing the directive have been initialized, Angular will respond.

```
import { Component, AfterViewInit, ViewChild, ElementRef } from '@angular/core';

@Component({
selector: 'app-example',
template: '<div #myDiv></div>',
})
export class ExampleComponent implements AfterViewInit {
@ViewChild('myDiv') myDiv!: ElementRef;

ngAfterViewInit(): void {
// Access and manipulate the DOM element here.
}
}
```

```
ngAfterViewInit(): void {
 // Access and manipulate the DOM element here.
 }
```

NgAfterViewChecked

The ngAfterViewChecked hook is called after every change detection cycle once the view and child views are checked. This can be utilized for performing extra actions once the view has been checked.

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

@Component({
selector: 'app-example',
template: '<p>{{ message }}</p>',
})
export class ExampleComponent implements AfterViewChecked {
message: string = '';

ngAfterViewChecked(): void {
// Additional actions after the view has been checked.
}
}
```

```
ngAfterViewChecked(): void {
 // Additional actions after the view has been checked.
 }
```

OnDestroy

Clean up just before Angular destroys the directive or component by unsubscribing Observables and detaching event handlers to prevent memory leaks.

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

@Component({
selector: 'app-counter',
template: `
<p>Current Count: {{ count }}</p>
<button (click)="startCounting()">Start Counting</button>
<button (click)="stopCounting()">Stop Counting</button>
`,
})
```

```
export class CounterComponent implements OnDestroy {
count: number = 0;
private countingSubscription: Subscription | undefined;

startCounting() {
// Simulate counting using an observable
const source = new Observable<number>((observer) => {
let value = 0;
const interval = setInterval(() => {
observer.next(value);
value++;
}, 1000);

// Cleanup when unsubscribed
return () => {
clearInterval(interval);
};
});

this.countingSubscription = source.subscribe((value) => {
this.count = value;
});
}

stopCounting() {
// Unsubscribe to prevent memory leaks
if (this.countingSubscription) {
this.countingSubscription.unsubscribe();
}
}

ngOnDestroy() {
// Ensure proper cleanup when the component is destroyed
this.stopCounting();
}
}
```