@Defer in Angular17

- Ashadeepa Debnath

ngPune Meetup - Saturday, Dec 9, 2023

About Me

- Based out of Tripura, brought up in Maharashtra, India
- BE in IT from RCEOM Nagpur University & PGDM-IT Management, Mumbai University
- Overall, 10 years of work experience in IT Industry
- Currently Working as a **Senior Software Architect** at Nice Interactive Solutions Ltd, Pune over 7 years
- My interests lie in Front End Engineering, and I love to architect, design and develop front-end applications
- I like to do canvas-painting & reading online articles in my leisure time

"My passion for knowledge-sharing & fostering collaboration drives me to continuously engage with fellow enthusiasts"

Email : ashadeepa.debnath@gmail.com
LinkedIn : https://www.linkedin.com/in/ashadeepa-debnath-16293575/

The Concept

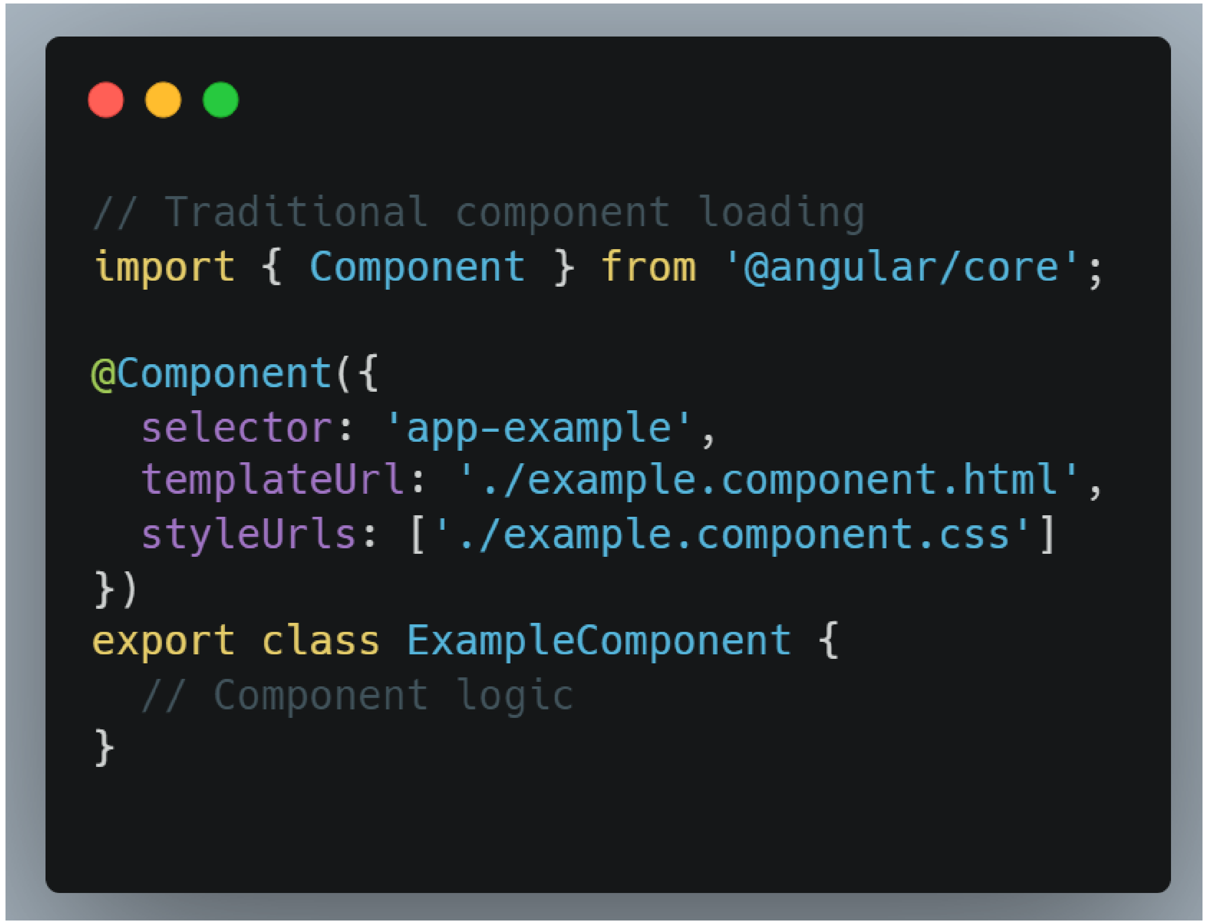
- With the latest control flow enhancements, Angular v17 introduces an impressive & highly beneficial feature : **the defer block**

- Here's a breakdown of the concept :
  - **Normal Script Execution**: By default, when a browser encounters a <script> tag, it stops parsing the HTML, executes the script, and then continues parsing the HTML. This can potentially delay the rendering of the page.
  - **Defer Attribute**: Adding the "defer" attribute to a <script> tag tells the browser to continue parsing the HTML while downloading the script in the background. The script will be executed only after the HTML parsing is complete

- It ensures that the HTML content is displayed to the user as soon as possible, and non-essential scripts are executed later

Traditional Component Loading in Angular

- Sequential loading of components during app initialization
- **Challenges**: Slower initial load, potential performance bottlenecks



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

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent {
  // Component logic
}
```

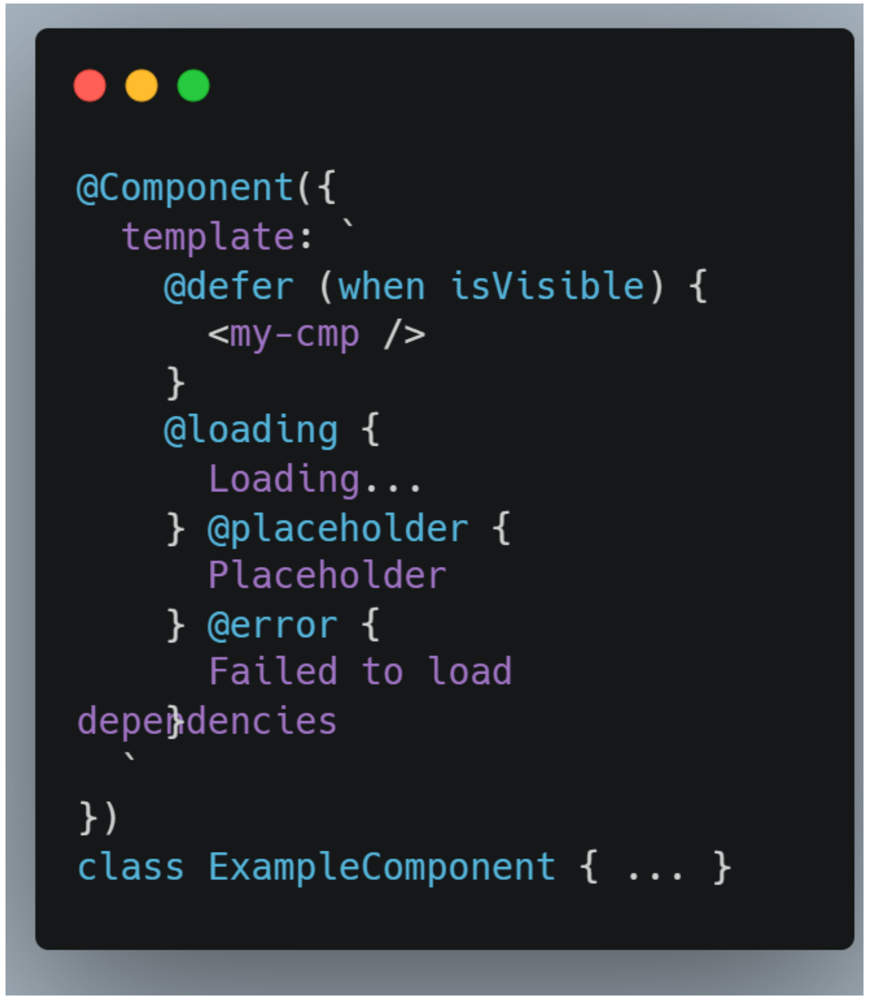
Introducing @defer

- Delaying component loading until needed & differentiate from traditional loading

- Deferrable views support a series of ***triggers***, ***prefetching***, & several sub ***blocks*** used for ***placeholder***, ***loading***, & ***error*** state management. You can also create custom conditions with ***when*** and ***prefetch when***.

  - **trigger** — defines when & how the component should be lazy loaded
  - **prefetch** — defines if, when & how the component lazy bundle should be pre-fetched

- **Benefits**: improved performance, optimized rendering



```
@Component({
  template: `
    @defer (when isVisible) {
      <my-cmp />
    }
    @loading {
      Loading...
    } @placeholder {
      Placeholder
    } @error {
      Failed to load
dependencies
    }
  `

})
class ExampleComponent { ... }
```

Requirements

Which dependencies are defer-loadable?

- **They must be standalone** & should be in its own **dedicated** file
- Component can only be used in the **parent template** (so not in **@ViewChild**, …)
- Transitive dependent components, directives and pipes used in the template of the deferred component can be both **standalone** and **ngModule** based

Defer Trigger Types

There are two types of @defer triggers

1. **on** (declarative) — uses one of the available behaviors (next slide)
2. **when** (imperative) — uses any custom logic that returns true or false (Component method, Signal, RxJs stream, etc.)

Defer Triggers

- **on interaction** — When the user clicks on the placeholder or another specified element
- **on timer** — Wait a predefined amount of time before fetching the component
- **on hover** — When the user hovers over the placeholder element, or another specific element
- **on immediate** — Retrieve the deferred chunk immediately
- **on idle** — When the browser has become idle and has stopped processing other tasks. We can achieve this by using the *requestIdleCallback* API
- **on viewport** — When the content is scrolled into view

Quiz : The @defer without any trigger will use the on ? by default

Defer Triggers

- **@defer (on <trigger>) {}** OR **@defer (when <logical expression >)** code creates a special block with a condition

Inside this @defer block, there are three optional blocks:

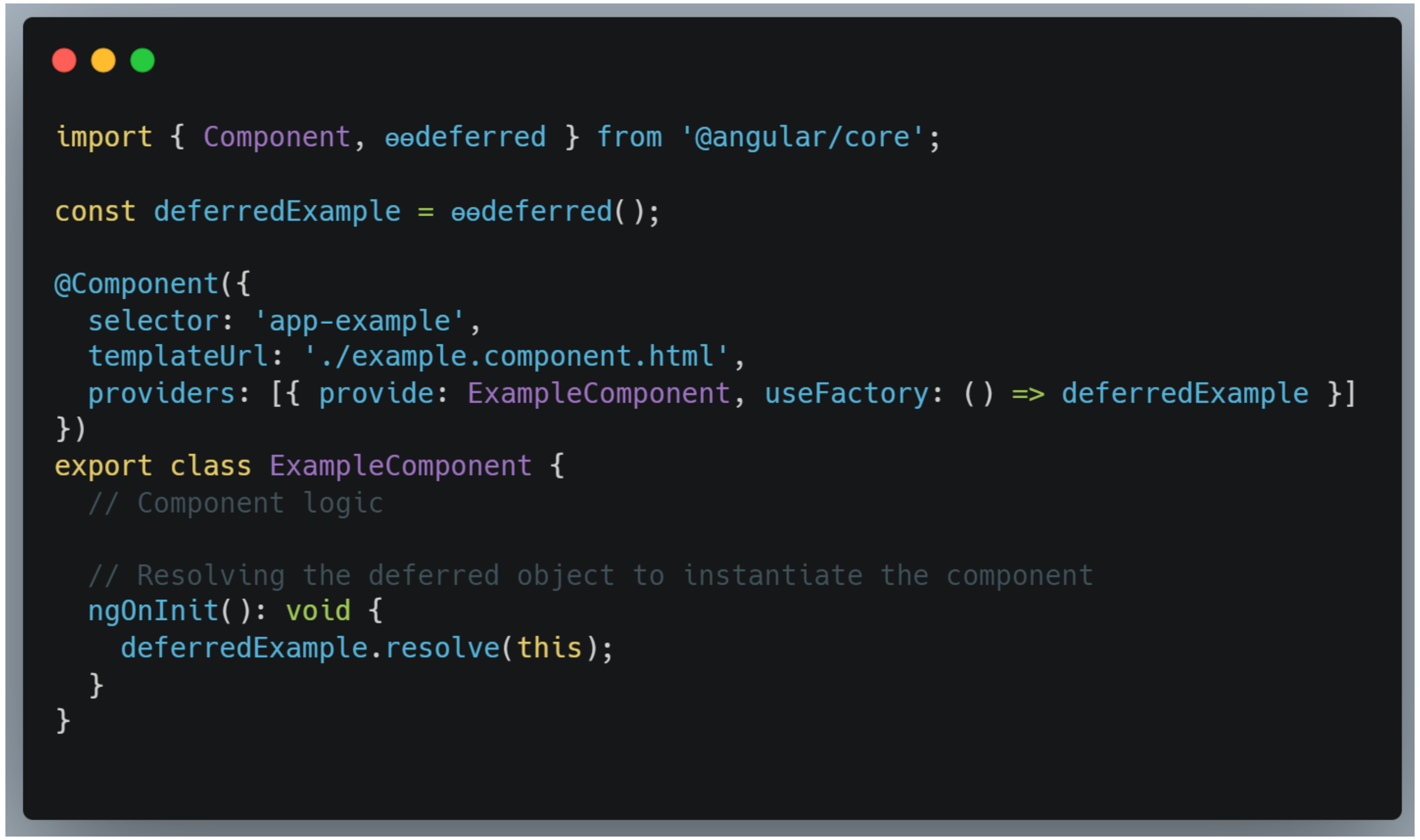
1. @**placeholder** block: Initially shown before the *@defer* condition is met
2. When the *@defer* condition is met, Angular loads content e.g., from the server, and you see the **@loading** block.
3. If loading fails, the **@error** block is displayed.

- **@defer (trigger1; trigger2; ... prefetch1; prefetch2; ...)** { } prefetch statements which belong to the @defer block… (remember, multiple statements are joined by the logical OR operator)

Simple Example

How @Defer Works

Under The Hood: Delaying instantiation until resolve

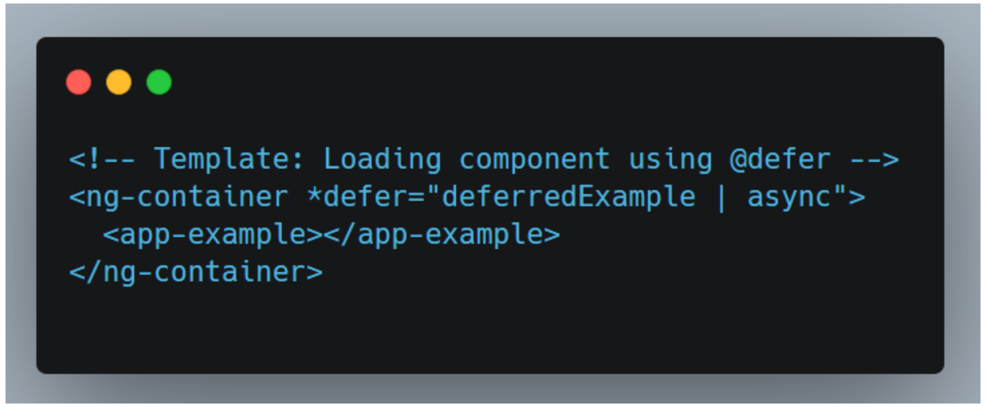


```
import { Component, ɵɵdeferred } from '@angular/core';

const deferredExample = ɵɵdeferred();

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  providers: [{ provide: ExampleComponent, useFactory: () => deferredExample }]
})
export class ExampleComponent {
  // Component logic

  // Resolving the deferred object to instantiate the component
  ngOnInit(): void {
    deferredExample.resolve(this);
  }
}
```



```
<!-- Template: Loading component using @defer -->
<ng-container *defer="deferredExample | async">
  <app-example></app-example>
</ng-container>
```

How @Defer Works

Under The Hood: Delaying instantiation until resolve

1. **Using ɵɵdeferred():** Angular provides the ɵɵdeferred() function to create a deferred object. This object represents the component to be loaded

2. **Service Injection with useFactory** : When defining the component, you use useFactory in the provider to delay its instantiation

3. **Template Usage**: In the parent component's template, you use *defer to handle the asynchronous resolution of the deferred component

4. **Delayed Instantiation**: When the parent component is rendered, the deferredExample observable waits for the ExampleComponent to be resolved before rendering it in the view

Best Practices

- **Identify Heavy Components**

  Determine components that significantly contribute to initial load times. These can be complex, resource-intensive components that aren't immediately necessary on app startup

- **Strategic Loading**

  Consider the user flow and prioritize loading components needed for the initial view. Defer loading less critical or secondary components that aren't required immediately

- **Utilize Lazy Loading**

  Leverage Angular' s lazy loading feature for modules and routes. Load modules or routes asynchronously when the user navigates to specific sections of the application

- **Testing and Performance Metrics**

  Measure the performance impact before and after implementation. Use tools like Chrome DevTools or Lighthouse to analyze loading times and assess improvements

Real-World Use cases

- **Optimizing Routes**

  Load route-specific components only when the user navigates to those routes. This ensures faster initial loading times for the main application view

- **Lazy Loading Heavy Components**

  Defer loading of heavy or less frequently used components until they are needed. For instance, charts, data grids, or advanced UI elements can be deferred if they aren't crucial for the initial view

- **User-Initiated Actions**

  Defer loading of heavy or less frequently used components until they are needed. For instance, charts, data grids, or advanced UI elements can be deferred if they aren't crucial for the initial view

Pitfalls to Avoid

- **Overusing Deferred Loading**
  Avoid deferring too many components unnecessarily. Loading too many components asynchronously might degrade the user experience as it introduces additional network requests and delays

- **Improper Handling of Dependencies**
  Ensure that dependencies required by deferred components are resolved correctly. Improper handling might lead to runtime errors or unexpected behavior

- **Complexity Over Clarity**
  Deferred loading can introduce complexity. Strive for a balance between performance optimization and code maintainability. Overly complex implementations might hinder code readability and maintenance

- **Not Considering User Experience**
  While optimizing for performance is essential, consider the impact on the user experience. Excessive loading delays or visible component loading might frustrate users

Conclusion

- Encourage everyone for implementation in your projects once you move to Angular17

- Resources for further exploration and learning
  - https://angular.dev/guide/defer#defer
  - https://angularexperts.io/blog/angular-defer-lazy-loading-total-guide
  - https://www.youtube.com/watch?v=i90lJ1qC-KE

Thank You

Ashadeepa Debnath