Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2020, 7(3):53-62

Review Article ISSN: 2394 - 658X

Angular Performance Best Practices

Yash Jani

Sr. Software Engineer
Fremont, California, US
yjani204@gmail.com

ABSTRACT

Angular [1] is a widely used framework for building dynamic single-page web applications [2]. However, ensuring optimal performance can be challenging as applications scale. This paper provides a detailed overview of best practices to enhance Angular [1] application performance. We explore key performance bottlenecks, such as inefficient Change detection [7] cycles and suboptimal DOM manipulations, and offer practical solutions [3]. The best practices discussed include efficient data binding techniques using the OnPush Change detection [7] strategy, optimized use of components and directives to minimize unnecessary DOM interactions, and designing reusable and efficient components. Additionally, we cover lazy loading and code splitting methods to reduce initial load times, and the judicious use of pipes and expressions to avoid performance degradation. By implementing these practices, developers can significantly improve the efficiency and responsiveness of Angular [1] applications.[4]

Key words: Angular [1], performance optimization, data binding, digest cycle, directive usage, component design, lazy loading, code splitting, filters, expressions, single-page application, web development, efficiency, responsiveness, front-end development, best practices.

–

INTRODUCTION

Since its release in 2016, Angular [1] has established itself as a robust framework for building dynamic, single-page web applications (SPAs). This framework, developed and maintained by Google, represents a complete rewrite of its predecessor, AngularJS, and introduces a wealth of new features and performance enhancements. Angular [1]'s component-based architecture, along with its reliance on TypeScript, offers developers a more modern and scalable approach to web development. However, maintaining optimal performance becomes a critical challenge as applications grow in complexity and size.

Angular [1] introduces fundamental concepts to enhance development efficiency and application performance. These include a powerful dependency injection system [5], an improved Change detection [7] mechanism, and support for reactive programming through RxJS [6]. Despite these advancements, developers often encounter performance bottlenecks, particularly when applications scale. Common issues include inefficient Change detection [7] cycles, suboptimal DOM manipulations, and improper use of Angular [1]’s extensive feature set. Addressing these issues requires a deep understanding of Angular [1]'s internal workings and adopting best practices tailored to the framework's architecture.

While the Change detection [7] mechanism in Angular [1] is more efficient than AngularJS's digest cycle, it can still pose significant performance challenges. Angular [1]'s default Change detection [7] strategy, which checks every component in the application tree for changes, can lead to performance degradation, especially in large applications. This paper explores the OnPush Change detection [7] strategy, which allows developers to optimize Change detection [7] by only checking components when their inputs change, thereby reducing the computational overhead.

Efficient use of Angular [1]'s component and directive system is another critical aspect of performance optimization. Components are the building blocks of Angular [1] applications, and their design directly impacts performance. Reusable and well-encapsulated components promote code maintainability and enhance runtime efficiency. On the other hand, directives provide a way to extend HTML functionality, but improper use can lead to excessive DOM manipulations and performance issues. This paper provides guidelines for creating performant components and directives, emphasizing the importance of minimizing unnecessary DOM interactions.

Lazy loading and code splitting are essential techniques for optimizing the initial load time of Angular [1] applications. By deferring the loading of non-critical modules and components until needed, developers can significantly reduce the initial payload, leading to faster application startup times. This paper discusses the lazy loading implementation using Angular [1]'s routing mechanism and explores the benefits of code splitting to improve application performance.

Angular [1]'s powerful template syntax, including pipes and expressions, offers a convenient way to transform and display data. However, improper use of these features can lead to performance degradation. Pipes, for instance, can be computationally expensive if used extensively in templates. This paper advocates for the judicious use of pipes and expressions, providing best practices for their efficient implementation.

Caching and memoization strategies are also explored to enhance performance. By storing the results of expensive computations and reusing them when needed, developers can reduce the need for redundant calculations and improve application responsiveness. This paper discusses implementing caching in Angular [1] using services and other built-in mechanisms.

Finally, continuous performance monitoring and profiling are essential for identifying and addressing performance bottlenecks. Tools such as Angular [1] DevTools and browser developer tools provide valuable insights into Angular [1] applications' performance characteristics. This paper emphasizes the importance of regular performance profiling and offers practical tips for using these tools to optimize Angular [1] applications.

In conclusion, this paper provides a comprehensive guide to optimizing Angular [1] applications. By implementing the best practices discussed, developers can address common performance issues and build Angular [1] applications that are both efficient and responsive. This exploration contributes to the broader discourse on web application performance and offers practical insights that are directly applicable to modern web development frameworks.

PROPOSED OPTIMIZATION TECHNIQUES

- **Efficient Change detection [7]**

Change detection [7] is one of the most critical aspects of Angular [1]'s performance. It is the process by which Angular [1] updates the view to reflect changes in the underlying data model. By default, Angular [1] uses the "default" Change detection [7] strategy, which checks every component in the application tree for changes, starting from the root component. While this ensures that the view is always up-to-date, it can become a performance bottleneck in large applications with many components, as it can result in unnecessary checks and updates.

1. **OnPush Change detection [7] Strategy**

The OnPush Change detection [7] strategy is a powerful optimization tool that developers can leverage to reduce the performance overhead associated with Angular [1]'s default Change detection [7] mechanism. When using OnPush, Angular [1] only checks a component for changes if one of its input properties changes, or if an event originates from the component or one of its children. This strategy significantly reduces the number of Change detection [7] cycles and the work Angular [1] has to perform.

```
// Example: Using OnPush Change detection
[7] Strategy
import { Component,
ChangeDetectionStrategy, Input } from
'@angular/core';

@Component({
  selector: 'app-onpush',
  template: `<p>{{ data }}</p>`,
  changeDetection:
ChangeDetectionStrategy.OnPush
})
export class OnPushComponent {
  @Input() data: string;
}
```

In this example, the `OnPushComponent` will only be checked for changes when the `data` input property changes. This means any changes to other application parts will not trigger Change detection [7] for this component, thus improving performance.

2. Immutable Data Structures

One key principle behind effectively using the OnPush Change detection [7] strategy is working with immutable data structures. Immutable data ensures that changes in data can be easily detected by Angular [1], as any change results in a new object reference. This is crucial for OnPush, as it relies on reference changes to determine if a component needs to be checked for updates.

```
// Example: Using Immutable Data
Structures
import { Component,
ChangeDetectionStrategy, Input } from
'@angular/core';

@Component({
  selector: 'app-immutable',
  template: `<p>{{ data.value }}</p>`,
  changeDetection:
ChangeDetectionStrategy.OnPush
})
export class ImmutableComponent {
  @Input() data: { value: string };
}

// Updating immutable data
const oldData = { value: 'Old Value' };
const newData = { ...oldData, value: 'New
Value' }; // New object reference
```

By ensuring that the `data` object is immutable, any change to its properties will result in a new object reference, which Angular [1] can detect and thus update the view accordingly.

3. Detaching and Reattaching Change detection [7]

Another advanced technique for optimizing Change detection [7] is manually detaching and reattaching change detectors. This is useful in scenarios where application parts are not frequently checked for changes.

```
// Example: Manually Detaching and
Reattaching Change detection
import { Component, ChangeDetectorRef }
from '@angular/core';
```

```
@Component({
  selector:
'app-manual-change-detection',
  template: `<p>{{ data }}</p>`
})
export class
ManualChangeDetectionComponent {
  data = 'Initial Data';

  constructor(private cdr:
ChangeDetectorRef) {
    // Detach Change detection
    this.cdr.detach();
  }

  updateData() {
    this.data = 'Updated Data';
    // Manually reattach Change detection
and check for changes
    this.cdr.detectChanges();
  }
}
```

In this example, the `ManualChangeDetectionComponent` has its change detector detached initially, meaning it will not participate in Angular [1]'s Change detection [7] mechanism. When the `updateData` method is called, the change detector is manually reattached and checked for changes using `detectChanges()`. This can significantly improve performance in scenarios where frequent updates are unnecessary.

4. Using TrackBy with NgFor

When using `ngFor` to render lists, the default Change detection [7] strategy can lead to performance issues. Angular [1] may re-render the entire list whenever the data changes. Using `trackBy` helps Angular [1] to identify items in the list, minimizing re-renders uniquely.

```
// Example: Using trackBy with NgFor
@Component({
  selector: 'app-track-by',
  template: `
    <ul>
      <li *ngFor="let item of items;
trackBy: trackByFn">{{ item.name }}</li>
    </ul>
  `
})
export class TrackByComponent {
  items = [{ id: 1, name: 'Item 1' }, {
id: 2, name: 'Item 2' }];

  trackByFn(index: number, item: any):
number {
    return item.id; // or item.id
  }
}
```

The `trackByFn` function allows Angular [1] to track each item by its unique identifier, reducing unnecessary DOM manipulations and enhancing performance.

- **Optimized Component and Directive Usage**

Optimizing the usage of components and directives is crucial for maintaining the performance and scalability of Angular [1] applications. Components and directives are the fundamental building blocks of Angular [1], and their design and implementation directly impact the efficiency of the application. Properly optimizing these elements can significantly improve performance and maintainability.

1. Reusable and Well-Encapsulated Components

Designing reusable and well-encapsulated components is essential for optimizing performance. Reusability enhances code maintainability and ensures that the same logic is not duplicated across different parts of the application, reducing redundancy. Encapsulation helps isolate the component's functionality, making it easier to manage and optimize.

```
// Example: Creating a Reusable Component
import { Component, Input } from
'@angular/core';
@Component({
  selector: 'app-reusable',
  template: `<div>{{ data }}</div>`
})
export class ReusableComponent {
  @Input() data: string;
}
```

In this example, the `ReusableComponent` is designed to display `data` passed to it via the data input property. This component can be reused across the application, ensuring the data display logic is centralized and easily manageable.

2. Avoiding Complex Logic in Templates

To maintain optimal performance, it is important to avoid placing complex logic in templates. Templates should focus primarily on displaying data and not contain extensive logic or computations. Instead, complex logic should be moved to the component class or services.

```
// Avoid complex logic in templates
@Component({
  selector: 'app-complex-logic',
  template: `<div>{{ processedData
}}</div>`
})
export class ComplexLogicComponent {
  @Input() data: string;
  processedData: string;

  ngOnInit() {
    this.processedData =
this.processData(this.data);
  }

  processData(data: string): string {
    // Complex logic here
    return data.toUpperCase();
  }}
```

The processData method contains the complex logic that processes the input data in this example. This approach ensures that the template remains simple and focused on data presentation.

3. Optimized Directive Implementation

Directives extend the functionality of HTML elements and are powerful tools in Angular [1]. However, improper use of directives can lead to performance issues, particularly due to excessive DOM manipulations. Optimizing directives involves using Angular [1]'s tools, such as isolate scope and proper linkage functions, to streamline DOM interactions and avoid unnecessary updates.

```
// Example: Creating a Performant
Directive
import { Directive, ElementRef, Input }
```

```
from '@angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input() set appHighlight(color:
string) {

this.el.nativeElement.style.backgroundCol
or = color;
  }
constructor(private el: ElementRef) {}
}
```

This `HighlightDirective` adds a background color to the host element. The directive efficiently updates the DOM element only when the input property changes, minimizing unnecessary DOM manipulations.

4. Utilizing Angular [1] Lifecycle Hooks

Angular [1] lifecycle hooks, such as `ngOnInit`, `ngOnChanges`, and `ngOnDestroy`, provide powerful mechanisms for optimizing component performance. These hooks allow developers to manage resources more effectively, initialize data at the appropriate times, and perform cleanup tasks to prevent memory leaks.

```
// Example: Using Lifecycle Hooks for
Optimization
import { Component, Input, OnChanges,
OnInit, OnDestroy } from '@angular/core';
@Component({
  selector: 'app-lifecycle',
  template: `<div>{{ data }}</div>`
})
export class LifecycleComponent
implements OnInit, OnChanges, OnDestroy {
  @Input() data: string;
  ngOnInit() {
    // Initialize data or start necessary
processes
    console.log('Component initialized');
  }
  ngOnChanges() {
    // Respond to changes in input
properties
    console.log('Input data changed');
  }
  ngOnDestroy() {
```

```
    // Clean up resources to prevent
memory leaks
    console.log('Component destroyed');
  }
}
```

In this example, the `LifecycleComponent` uses `ngOnInit` to initialize data, `ngOnChanges` to react to input property changes, and `ngOnDestroy` to perform cleanup tasks. These hooks help manage the component's lifecycle efficiently, ensuring optimal performance and resource utilization.

5. Minimizing Change detection [7] Impact

In addition to the OnPush Change detection [7] strategy, developers can further minimize the impact of Change detection [7] by optimizing the structure of their components and directives. For example, breaking down complex components into smaller, more focused components can reduce the scope of Change detection [7], making the application more efficient.

```
// Example: Breaking Down Complex
Components
import { Component, Input } from
'@angular/core';

@Component({
  selector: 'app-parent',
  template: `
    <app-child
[childData]="data"></app-child>
  `
})
export class ParentComponent {
  data: string = 'Parent Data';
}

@Component({
  selector: 'app-child',
  template: `<div>{{ childData }}</div>`
})
export class ChildComponent {
  @Input() childData: string;
}
```

In this example, the complex logic is split between the `ParentComponent` and the `ChildComponent`, reducing the Change detection [7] scope and improving performance.

6. Avoiding Unnecessary DOM Manipulations

Frequent and unnecessary DOM manipulations can degrade performance. Directives should be designed to minimize DOM manipulations by leveraging Angular [1]'s built-in features and best practices. For example, instead of directly manipulating the DOM within a directive, consider using Angular [1]'s built-in structural directives like `ngIf` and `ngFor` to add or remove elements from the DOM conditionally.

```
// Example: Avoiding Unnecessary DOM
Manipulations
import { Component } from
'@angular/core';

@Component({
  selector: 'app-structural',
  template: `
    <div *ngIf="isVisible">This is
conditionally visible</div>
    <ul>
      <li *ngFor="let item of items">{{
item }}</li>
    </ul>
  `
})
export class StructuralComponent {
```

In this example, ngIf and ngFor are used to conditionally display elements and iterate over a list, respectively, reducing the need for manual DOM manipulations and ensuring more efficient rendering.

Leveraging Angular [1]'s Reactive Forms

Angular [1]'s reactive forms provide a powerful and efficient way to manage form inputs, validation, and submission. Reactive forms are more performant than template-driven ones because they use an explicit and immutable approach to managing form states.

```
// Example: Using Reactive Forms
import { Component, OnInit } from
'@angular/core';
import { FormBuilder, FormGroup } from
'@angular/forms';

@Component({
  selector: 'app-reactive-form',
  template: `
    <form [formGroup]="form">
      <label>
        Name:
        <input formControlName="name">
      </label>
      <button
type="submit">Submit</button>
    </form>
    <div *ngIf="form.valid">Form is
valid!</div>
  `
})
export class ReactiveFormComponent
implements OnInit {
  form: FormGroup;

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.form = this.fb.group({
      name: ['']
    });
  }
```

In this example, the ReactiveFormComponent uses Angular [1]'s reactive forms to manage the form state. This approach ensures that the form state is managed more predictably and efficiently, reducing unnecessary re-renders and enhancing performance.

- **Lazy Loading and Code Splitting**

Lazy loading and code splitting are advanced optimization techniques in Angular [1] applications that enhance performance by reducing initial load times and improving responsiveness. Lazy loading leverages Angular [1]'s routing mechanism to load non-essential modules only when required, thereby minimizing the initial bundle size and speeding up the application's initial render. This is achieved by dynamically configuring routes to load modules using Angular [1]'s loadChildren syntax. Code splitting, often facilitated by modern build tools like Webpack, breaks the application into smaller, asynchronously loaded chunks based on usage patterns. This means that only the necessary code for the current view is fetched and executed, reducing unnecessary resource consumption and improving application performance. Together, these techniques optimize resource utilization, decrease network load, and make large-scale applications more maintainable and scalable by isolating features and dependencies into distinct, on-demand modules. This strategic loading approach ensures a smoother, faster user experience and a more efficient development workflow.

- **Efficient Use of Pipes and Expressions**

Efficient use of pipes and expressions [8] is critical for optimizing Angular [1] application performance by minimizing unnecessary calculations and updates. Pipes transform data in templates and can be either pure or impure. Pure pipes are highly efficient as they execute only when input data changes, making them suitable for most scenarios. Impure pipes, on the other hand, run on every Change detection [7] cycle, which can lead to performance bottlenecks if used excessively. Therefore, it's important to use impure pipes sparingly. Placing complex logic within Angular [1] expressions can degrade performance, as these expressions are evaluated frequently during Change detection [7]. Instead, move complex logic to component classes or services where it can be managed more efficiently. This approach optimizes the performance and keeps the templates clean and focused on data presentation. By strategically using pure pipes and minimizing complex expressions in templates, developers can significantly reduce the computational load and enhance the responsiveness of Angular [1] applications.

Example

```
// pure-uppercase.pipe.ts
@Pipe({name: 'pureUppercase', pure:
true})
export class PureUppercasePipe implements
PipeTransform {
  transform(value: string): string {
    return value.toUpperCase();
  }
}

// my-component.component.ts
@Component({
  selector: 'app-my-component',
  template: `<div>{{ message |
pureUppercase }}</div>`
})
```

```
export class MyComponent {
  message: string = 'hello world';
}
```

In this example, the PureUppercasePipe is a pure pipe that efficiently transforms a string to uppercase only when the input changes. The component uses this pipe in the template, ensuring minimal performance impact.

CONCLUSION

In this paper, we explored various techniques to optimize the performance of Angular [1] applications, focusing on efficient use of pipes and expressions, lazy loading and code splitting, optimized component and directive usage, and efficient Change detection [7]. These strategies collectively enhance Angular [1] applications' scalability, maintainability, and responsiveness.

Efficient use of pipes and expressions helps to minimize unnecessary calculations and updates. By utilizing pure pipes and avoiding complex logic within templates, developers can reduce the computational overhead associated with Change detection [7] cycles. This keeps the templates clean and focused on data presentation, improving overall application performance.

Lazy loading and code splitting are essential for reducing initial load times and improving application responsiveness. By loading non-essential modules and components only when needed, these techniques optimize resource utilization and enhance user experience. Strategic use of route-based and component-based splitting ensures that applications remain efficient and maintainable as they grow in size and complexity.

Optimized component and directive usage is crucial for maintaining performance in large-scale Angular [1] applications. Designing reusable and well-encapsulated components, avoiding complex logic in templates, and implementing efficient directives help to streamline DOM interactions and reduce unnecessary updates. Leveraging Angular [1]'s lifecycle hooks and minimizing Change detection [7] impact further improve performance.

Efficient Change detection [7] is fundamental to Angular [1]'s performance. Adopting the OnPush Change detection [7] strategy, using immutable data structures, and manually detaching and reattaching change detectors where appropriate can significantly reduce the computational load. These practices ensure the application remains responsive and performs well even as it scales.

By implementing these best practices, developers can create Angular [1] applications that are not only high-performing but also scalable and maintainable. These optimizations provide a smoother and faster user experience, leading to more robust and efficient web applications.

REFERENCES

[1]. “Angular”. https://angular.dev/.

[2]. Authors, “AngularJS in the Wild: A Survey with 460 Developers | Request PDF”. https://www.researchgate.net/publication/30 9368617_AngularJS_in_the_Wild_A_Surve y_with_460_Developers.

[3]. “Performance Tuning Angular [1] Apps – From 0 to 100 (live talk)”. https://christianlydemann.com/perfor mance-tuning-angular-apps-from-0-to-100-li ve-talk/

[4]. “Angular Tools for High Performance”. https://blog.angular.io/angular-tools-for-high-performance-6e10fb9a0f4a? gi=f43f58b49848

[5]. “Dependency Injection in Angular”. https://blog.thoughtram.io/angular/ 2015/05/18/dependency-injection-in-angular-2.html

[6]. “RxJS concepts for dummies” https://medium.com/@nkchandupatla/reacti ve-programming-and-rxjs-concepts-in-4-min s-e0b1bb142360

[7]. “Change detection” https://trepo.tuni.fi/bitstream/handle/123456 789/26217/Hakulinen.pdf

[8]. “Angular Performance: Optimizing Expression Re-evaluation with Pure Pipes” https://blog.bitsrc.io/angular-performanc e-optimizing-expression-re-evaluation- with-pure-pipes-ff8df36ed478