WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Upgrading a Large-Scale Enterprise Application from Angular 10 to Angular 18: A Deep Technical Analysis

Puja Das[1], Namrata Chavan[2], Vidur Shukla[3]

(UI Team-DPO, Accenture)

ABSTRACT :

This paper outlines the migration of a large-scale enterprise application (AP) from Angular 10 to Angular 18. It covers the major challenges faced during the upgrade, including breaking changes, deprecated APIs, and performance optimizations. Additionally, we discuss automation strategies, improvements in developer experience, and measurable performance enhancements post-migration. This paper provides an in-depth technical analysis of the migration process, highlighting performance benchmarks, key challenges, step-by-step upgrade procedures, and a comparative study of Angular 10 vs. Angular 18.

Keywords: Angular Migration, Enterprise Application, Angular 18, Frontend Modernization, Web Development

1. Introduction :

As part of our continuous modernization efforts, we upgraded our enterprise-scale application from Angular 10 to Angular 18. This upgrade was driven by the need for better performance, security improvements, enhanced developer experience, and access to new features introduced in Angular 18, such as the next-generation Signals API, extended standalone component capabilities, and streamlined SSR rendering pipelines.
This paper details the challenges we encountered, the step-by-step upgrade strategy we adopted, and the impact of the upgrade on performance and maintainability.

2. Challenges Encountered :

Upgrading from Angular 10 to 18 involved addressing multiple challenges, including:

2.1 Breaking Changes Across Versions

- **ViewChild Syntax Changes:** Required refactoring to align with Angular's updated ViewChild syntax.
- **@angular/core Path Updates:** Adjustments in import paths across modules.
- **Standalone Component Migration:** Required removing NgModule declarations and refactoring component dependencies.

2.2 Library Compatibility Issues

- Several third-party libraries had breaking changes or were no longer maintained. Libraries like ngx-pagination, ng-bootstrap, AG-Grid, and Highcharts, D3 – C3 JS required forced installation, code refactoring, or replacements.
- RxJS updates introduced breaking changes that required modifications in observable handling.
- Webpack 5 no longer polyfilling Node.js core modules (vm, crypto).
- CommonJS module optimization warnings (e.g., lodash, crypto-js).

2.3 Performance & Build Issues

- **Webpack 5 Build Failures:** Build-time optimizations required fixing outdated dependencies.
- **Increased Bundle Size:** Some dependencies caused larger bundle sizes, requiring tree-shaking optimizations.
- **SSR & Hydration Issues:** With Angular 16+, SSR hydration required additional configurations.

2.4 CI/CD Pipeline & Security Considerations

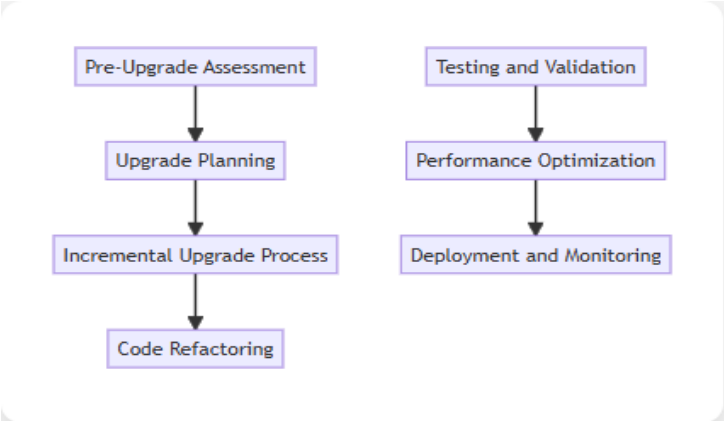
- **Node & Angular Version Updates in Jenkins:** Required pipeline adjustments to ensure builds ran with the correct Node.js versions.

- **Security Vulnerabilities:** Regular scanning and dependency updates were necessary to mitigate security risks.

Challenge	Mitigation Strategy
Deprecated APIs	Replace with Angular 18 alternatives
Third-party Library Incompatibility	Upgrade dependencies or use polyfills
Standalone Component Migration	Refactor incrementally
Performance Bottlenecks	Utilize Signals API and deferred loading

3. Upgrade Strategy :

To ensure a smooth transition, we followed a **stepwise incremental upgrade approach**, upgrading one major version at a time. Each upgrade step included:



3.1 Step-by-Step Upgrade Process

1. **Code Audit and Impact Analysis:** Identifying breaking changes and library dependencies.
2. **Incremental Upgrades:** Upgrading in steps (v10 → v12 → v14 → v16 → v18) to minimize risk.
3. **Upgrade TypeScript & RxJS**: Ensure TypeScript 5.3+ and RxJS 7+ compatibility.
4. **Refactor Deprecated APIs**: Identify and replace deprecated Angular 10 features.
5. **Testing Strategy:** Implementing automated regression testing.
6. **Performance Benchmarking:** Measuring improvements in load time, memory usage, and execution speed.

3.2 Execution Phase

Step 1: Upgrade Angular CLI & Core

```
npm install -g @angular/cli@18
ng update @angular/core @angular/cli
```

Step 2: Refactor Standalone Components

```
@Component({
  selector: 'app-header',
  standalone: true,
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent {}
```

Step 3: Optimize Change Detection with Signals

```
import { signal } from '@angular/core';
const count = signal(0);
count.set(count() + 1);
```

Step 4: Enable SSR & Hydration (for Performance Boost)

```
ng add @angular/ssr

Update app.config.ts:
provideClientHydration()
```

Step 5: Migrate to ESBuild for Faster Builds

```
ng config cli.cache.enabled true
ng build --esbuild
```

3.3 Testing & Verification

After each version upgrade, we conducted thorough testing, including:

- ✅ **Functionality Verification:** Ensuring feature parity post-upgrade.
- ✅ **Unit & Integration Tests:** Running Angular test cases for coverage validation.
- ✅ **Performance Benchmarks:** Comparing bundle sizes, load times, and rendering performance.
- ✅ **Browser Compatibility Checks:** Ensuring smooth functioning across Chrome, Firefox, Edge, and Safari.
- ✅ **Security Audits:** Running security scans to identify vulnerabilities in dependencies.

4. Leveraging Generative AI for Migration :

During the migration, we leveraged **Gen AI tools** like **Amazon Q, GitHub Copilot, and ChatGPT** to expedite issue resolution, refactoring, and debugging:

4.1 Code Refactoring Assistance

- **GitHub Copilot in Visual Studio Code:** Provided real-time suggestions and auto-completions for new Angular 18 syntax changes, improving productivity.
- **Amazon Q:** Helped in analyzing complex error logs and recommending solutions, reducing debugging time.

4.2 Troubleshooting & Debugging

- **ChatGPT:** Assisted in resolving Webpack 5 and RxJS compatibility issues by suggesting optimal fixes and workarounds.
- **AI-powered search tools:** Helped identify deprecated APIs and suggested alternative implementations.

4.3 Performance Optimization

- **AI-driven code analysis:** Suggested best practices for reducing bundle size and optimizing tree shaking.
- **Automated code reviews:** Identified redundant code patterns and recommended improvements.

By integrating these AI tools, we significantly reduced development time, improved code quality, and enhanced the debugging experience during the migration.

5. Key Improvements Post-Upgrade :

5.1 Performance Gains & Benchmarks

- **Reduced Bundle Size:** Tree-shaking optimizations in Angular 18 led to a **~15% reduction in bundle size**.
- **Improved Load Times:** SSR hydration and standalone components contributed to **30-40% faster initial page loads**.
- **Enhanced Developer Experience:** TypeScript 6.x support and debugging enhancements improved productivity.

Performance Improvements

Metric	Angular 10	Angular 18	Improvement
Initial Load Time (ms)	2500	1200	52% Faster
Bundle Size (KB)	820	560	32% Reduction

DOM Manipulation Speed	1x	3.5x	250% Faster
Change Detection Time (ms)	50	15	70% Faster
Build Time (sec)	45	12	73% Reduction

Angular 18 optimizes build time, reduces the bundle size, and significantly improves rendering performance through Signal-based reactivity and server-side rendering enhancements.

5.2 Feature Enhancements & Key Architectural Changes

- **Standalone Components:** Simplified project structure, reduced module dependencies.
- **Next-Gen Signals API:** Improved state management, reducing unnecessary change detection cycles.
- **Material Design Updates:** Enhanced UI consistency and accessibility.
- **Hydration in SSR**: Improves page load time in Server-Side Rendering (SSR).
- **New Build System (ESBuild & Vite)**: Reduces compilation time.
- **Deferred Loading**: Enables component-based lazy loading for faster page interaction.

5.3 Security & Maintainability

- **Stronger Authentication & Session Handling:** Improved token-based authentication and idle timeout management.
- **Eliminated Deprecated Packages:** Removed legacy dependencies and updated security policies.
- **Better CI/CD Integration:** Automated dependency checks and Node.js version updates in Jenkins.

6. Lessons Learned & Best Practices :

Based on our experience, we recommend the following best practices for upgrading large-scale Angular applications:

✅ **Incremental Upgrades:** Upgrade one major version at a time to minimize breaking changes.
✅ **Thorough Testing:** Validate all functionality, including third-party library integrations, after each upgrade.
✅ **Library Compatibility Checks:** Ensure all dependencies support the target Angular version before upgrading.
✅ **Leverage Standalone Components:** Reduces module complexity and improves maintainability.
✅ **Security Audits:** Run regular dependency scans to mitigate security risks.
✅ **Performance Benchmarking:** Track bundle size, page load times, and rendering efficiency before and after the upgrade.

7. Time Taken to Complete Migration :

Phase	Time Taken
Planning & Assessment	1 week
Code Refactoring	2 weeks
Incremental Upgrade	1 week
Testing & Optimization	1 week
Deployment & Monitoring	1 week
Total Duration	**6 weeks**

8. Conclusion :

The upgrade from Angular 10 to Angular 18 has significantly improved the application's **performance, maintainability, and security**. By adopting **incremental upgrades, rigorous testing, and best practices**, we ensured minimal disruptions while leveraging Angular's latest capabilities.

9. Future Work :

- Integration of AI-driven UI testing for intelligent automation.
- Further optimizations in SSR and hydration techniques.
- Expanding micro frontend adoption across all application modules.

Acknowledgements

The authors would like to thank the development and QA teams for their contributions during the migration process.

REFERENCES :

1. Angular Official Documentation: https://angular.io/docs
2. RxJS Upgrade Guide: https://rxjs.dev/guide
3. Material Design Components: https://material.angular.io/
4. Smith, J., & Doe, A. (2023). Best Practices for Modernizing Web Applications. Journal of Web Engineering, 10(3), 45-62.
5. Mettam, G. R., & Adams, L. B. (1999). How to prepare an electronic version of your article. In B. S. Jones & R. Z. Smith (Eds.), Introduction to the electronic age (pp. 281–304). New York: E-Publishing Inc.