**Problem Statement:**

LAZ currently maintains a complex ecosystem of 21 AngularJS 1.6 applications, all of which are hybrid applications built with Angular 18 using the Angular UpgradeModule. While these applications have benefited from the introduction of Angular 18 components and TypeScript support, the continued reliance on AngularJS presents several challenges:

* **Technical Debt:** The AngularJS codebase may be accumulating technical debt due to its outdated nature and potential compatibility issues.
* **Maintenance Costs:** Maintaining and supporting AngularJS applications can be resource-intensive and time-consuming.
* **Security Risks:** Older frameworks like AngularJS may be more vulnerable to security vulnerabilities.
* **Developer Experience:** Modern development practices and tooling may be hindered by the use of AngularJS.
* **Future-Proofing:** Continued reliance on AngularJS may limit the organization's ability to adopt newer technologies and frameworks.

To address these challenges and ensure the long-term sustainability of our applications, LAZ seeks a partner to assist in migrating the AngularJS applications to a fully Angular 18-based architecture. The migration should prioritize a seamless transition, minimizing disruptions to our business operations and preserving existing functionalities.

**Key Objectives:**

* Modernize AngularJS applications to leverage the full potential of Angular 18.
* Improve application performance, scalability, and maintainability.
* Enhance developer productivity and reduce maintenance costs.
* Mitigate security risks associated with AngularJS.
* Ensure a smooth transition with minimal disruptions to business operations.

**Desired Outcomes:**

* Fully migrated Angular 18 applications with clean, well-structured codebases.
* Improved application performance and user experience.
* Enhanced security and compliance.
* Reduced technical debt and maintenance costs.
* A future-proof application architecture that can easily adapt to new technologies and trends.

**Angular Upgrade with Hybrid Approach:**

**Description:** This approach involves a gradual migration, combining elements of incremental upgrades and greenfield redevelopment. It's suitable for complex applications or when specific areas require a complete overhaul.

**Key Benefits:**

* **Balanced risk:** Minimizes the risk associated with a complete rewrite.
* **Leverages existing code:** Reuses parts of the existing application, saving time and effort.
* **Modernization:** Introduces new features and improvements in specific areas.
* **Flexibility:** Adapts to different application complexities and business requirements.

**Considerations:**

* **Increased complexity:** Requires careful planning and coordination between teams.
* **Potential for inconsistencies:** May introduce inconsistencies between upgraded and newly developed parts.
* **Technical debt:** If not managed carefully, can introduce technical debt.

**Complete Rewrite (Big Bang Approach):**

* **Description:** This approach involves creating entirely new Angular 18 applications from scratch, transferring essential logic and resources from the existing AngularJS applications.
* **Key Benefits:**
  + **Clean codebase:** A fresh start allows for a well-structured and maintainable codebase.
  + **Modern architecture:** Leverages the latest Angular features and best practices.
  + **Performance improvements:** Potentially achieves better performance through optimization techniques.
  + **Reduced technical debt:** Avoids carrying over legacy issues from AngularJS.
* **Considerations:**
  + **Potential for data migration issues:** Data migration can be complex and error-prone.
  + **Risk of feature omissions:** There's a risk of overlooking or omitting features from the existing applications.

**Case Study 1:**

Application Modernization through Angular Upgrade

**1. Client Overview**

* **Industry:** Cloud Security
* **Application Type:** Cloud based SASS Platform
* **Challenges:**
  + 8 different Angular apps, having dependency on outdated dependencies
  + Vulnerable to security vulnerabilities.
  + Different applications having different methodologies and programming approaches leading to inconsistencies,
  + Obsolete and Redundant Code within each application
  + Harder to maintain and scale

### **2. Project Goals**

* **Objectives:**
  + Upgrade the AngularJS applications to Angular 14.
  + Improve application performance, scalability, and maintainability.
  + Enhance developer productivity and reduce maintenance costs.
  + Mitigate security risks associated with AngularJS.
  + Ensure a smooth transition with minimal disruptions to business operations.
* **Scope:**
  + Upgrade all the 8 different applications
  + Build a design system using a story book and create a comprehensive documentation.
  + Migrate the reusable components from each app to design system
  + Refactor the code and integrate the NEO components
  + Create a single workspace to manage all the angular applications

### **3. Approach and Methodology**

* **Strategy:** Big Bang Approach
* **Tools and Technologies:** For this migration project, we utilized Nx for project generation and management, Jest for unit testing. We also leveraged Webpack for bundling and optimization, and ESLint for code quality checks.
* **What were the Challenges:**
  + **Inconsistent Codebases:** Different teams adopted varying coding practices and standards, leading to inconsistent codebases.
  + **Dependency Management:** Managing dependencies across multiple applications was complex, especially when dealing with version conflicts.
  + **Integration with SaaS Platform:** Ensuring seamless integration with the existing SaaS platform while making significant changes to the applications was challenging.
  + **Team Coordination:** Coordinating efforts across different teams to ensure a smooth migration was difficult, especially when dealing with conflicting priorities.
* **How we addressed those challenges:**
  + **Inconsistent Codebases:** Implementing a linting tool to enforce consistent coding standards across all applications.
  + **Dependency Management:** We leverage a dependency management tool like Nx to manage dependencies and resolve conflicts.
  + **Integration Testing:** Creating automated integration tests to verify the interaction between the migrated applications and the SaaS platform.
  + **Team Coordination:** Establishing regular team meetings and using project management tools to track progress and coordinate efforts.

**4. Results and Outcomes**

* Reduced development time across 8 different teams by at least 30%
* Improved application performance from 30 to 91
* Best Practices score was improved from 44 to 96

**Case Study 2:**

**1. Client Overview**

* **Industry:** EduTech
* **Application Type:** LMS
* **Challenges:**
  + AngularJS app, having dependency on outdated dependencies
  + Having security vulnerabilities due to outdated and unsupported libraries/framework.
  + Obsolete and Redundant Code.
  + Harder to maintain and scale

### **2. Project Goals**

* **Objectives:**
  + Upgrade the AngularJS application to Angular 18.
  + Improve application performance, scalability, and maintainability.
  + Enhance developer productivity and reduce maintenance costs.
  + Mitigate security risks associated with AngularJS.
  + Ensure a smooth transition with minimal disruptions to business operations.
* **Scope:**
  + Upgrade existing AngularJS application
* Use TailwindCSS and Angular Styleguide to rewrite code.
* Migrate the reusable components from JavaScript to TypeScript.
* Refactor the code and integrate the Inhouse library.

### **3. Approach and Methodology**

* **Strategy:** Agile Methodology
* **Tools and Technologies:** For this migration project, we utilized Github copilot, Jasmine/Karma for unit testing. We also leveraged Webpack for bundling and optimization, and ESLint for code quality checks.
* **What were the Challenges:**
  + **Inconsistent Codebases: Different teams adopted varying coding practices and standards, leading to inconsistent codebase.**
  + **Team Coordination: Coordinating efforts across different locations to ensure a smooth migration was difficult, especially when dealing with conflicting priorities.**

* How we addressed those challenges:
  + **Inconsistent Codebases:** Implementing a linting tool to enforce consistent coding standards across all applications.
  + **Team Coordination:** Establishing regular team meetings and using project management tools to track progress and coordinate efforts.

**4. Results and Outcomes**

* Reduced development time by using Github copilot by at least 15%
* Improved application performance
* Best Practices score was improved

**Proposed Solution:**

**Leveraging Existing Hybrid Approach: UpgradeModule Strategy**

Building upon the existing hybrid applications, which utilize Angular 18 components within AngularJS 1.6 apps, we propose a migration approach centered on the **UpgradeModule**. This approach capitalizes on the groundwork laid by the hybrid approach while taking advantage of the UpgradeModule for a more streamlined transition.

**Benefits of UpgradeModule:**

* **Incremental Upgrades:** The UpgradeModule allows for a gradual upgrade process, minimizing disruptions and enabling focused efforts on specific components.
* **Coexistence:** Existing AngularJS code and upgraded Angular components can coexist within the same application, ensuring a smooth transition.
* **Maintainability:** Code changes are contained within upgraded components, simplifying future maintenance efforts.
* **Reduced Risk:** By focusing on targeted upgrades, the overall risk associated with the migration is minimized.

**Proposed Workflow:**

1. **Assessment and Prioritization:** We will collaborate with the technical teams to identify critical components or functionalities that benefit most from an upgrade.
2. **Upgrade Selected Components:** Using the UpgradeModule, we will gradually upgrade the identified components to Angular 18, leveraging existing hybrid application structure.
3. **Testing and Integration:** After each upgrade, we will conduct thorough testing to ensure compatibility and seamless integration with the existing application and SaaS platform.

**Streamlining Angular Upgrades with a Monorepo.**

Managing 21 separate Angular applications across different GitHub repositories is challenging. This fragmented approach makes coordination, dependency management, and future upgrades complex and time-consuming. Therefore, we additionally propose implementing a Mono repo strategy using Nx Workspace tool to consolidate all the applications into a single repository. This centralized approach can provide significant benefits including:

* **Improved Coordination:** By bringing all applications under a single umbrella, we can streamline communication and collaboration among development teams.
* **Simplified Dependency Management:** Nx's built-in dependency management capabilities can ensure consistent versions across applications, reducing conflicts and errors.
* **Streamlined Upgrades:** The monorepo structure can facilitate efficient upgrades to Angular 18, as changes could be applied consistently across all applications.
* **Shared Resources:** We can share common resources like libraries, components, and utilities across applications, promoting code reusability and reducing redundancy.

**Streamlining Angular Upgrades with Micro-Frontends:**

This is an alternative to monorepo architecture where each angular app would be converted to a micro-frontend and integrates via Module Federation. This approach will also provide several benefits such as:

* **Decentralized Development:** Each micro-frontend could be developed and managed independently by different teams, fostering autonomy and flexibility.
* **Scalability:** The micro-frontend architecture allows for easier scaling and maintenance of individual applications.
* **Independent Upgrades:** Each micro-frontend could be upgraded and deployed independently, reducing the risk of downtime, and allowing for a more agile development process.
* **Reduced Coupling:** The loose coupling between micro-frontends minimizes the impact of changes in one application on others, reducing the risk of integration issues.