# Application Evergreening for Java-Based Applications

Introduction  
Application evergreening is an essential practice for keeping Java-based applications updated with the latest dependency   
versions and security patches. In a complex Java ecosystem, applications often rely on multiple parent and child   
dependencies, and failure to upgrade these can introduce vulnerabilities, cause performance issues, or make the application   
incompatible with newer libraries or technologies. Evergreening ensures that an application remains current by   
automatically resolving, updating, and testing its dependencies. This case study presents an approach for automating   
the evergreening process using Jenkins, covering key stages, constraints, architecture, and conclusions.  
  
Approach Overview  
The evergreening process ensures that all dependencies, both parent and child, are automatically upgraded while minimizing   
manual intervention. The primary focus of this approach is to:  
- Ensure the latest stable versions of dependencies are used.  
- Fix vulnerabilities associated with outdated dependencies.  
- Ensure stability by testing the application after each upgrade.  
- Simplify the deployment process to a Platform-as-a-Service (PaaS) environment.  
  
Workflow (Stages)  
  
Stage 1: Get Input Request Parameters  
The pipeline begins by receiving parameters that guide which dependencies should be included, excluded, or need special   
treatment during the upgrade.  
  
Stage 2: Checkout Git  
The application's source code is checked out from the version control system (e.g., Git) to retrieve the current state of   
the codebase along with its dependencies.  
  
Stage 3: Identify Dependencies  
The pipeline identifies both parent and child dependencies within the project (e.g., in pom.xml for Maven or build.gradle   
for Gradle) to map the dependency graph.  
  
Stage 4: Upgrade Spring Parent Dependencies  
The first step in upgrading focuses on Spring parent dependencies. These are foundational for Spring-based applications   
and need to be up-to-date to ensure compatibility with other libraries.  
  
Stage 5: Exclude Specified Dependencies  
Any dependencies that need to be excluded from the upgrade process (based on the input request) are skipped. This could   
include legacy dependencies that may require specific versions to maintain application stability.  
  
Stage 6: Upgrade Common Dependencies  
Next, the pipeline upgrades any common dependencies that are not specific to the Spring framework but are required for   
application functionality (e.g., database drivers, utility libraries).  
  
Stage 7: Run Individual Dependency Tests  
After each dependency is upgraded, individual unit and integration tests are executed to ensure the new version does not   
break the application.  
  
Stage 8: Iterative Dependency Upgrade  
The system iterates through the dependencies, upgrading from the latest (highest) versions to older ones if needed, until   
the test scripts pass.  
  
Stage 9: aPaaS Deployment  
Once all upgrades are complete and tests pass, the application is deployed to an aPaaS (application Platform-as-a-Service)   
environment for staging.  
  
Stage 10: Prisma Scan  
A Prisma security scan is conducted to detect any vulnerabilities in the new dependency versions or in the application   
code.  
  
Stage 11: Create Pull Request  
If the upgrades are successful, the pipeline creates a pull request in the version control system, outlining the changes   
made. This pull request can then be reviewed before being merged into the main branch.  
  
Architecture  
The architecture for this evergreening pipeline consists of several core components:  
1. Jenkins Pipeline: A Jenkins pipeline orchestrates the entire process, from pulling the code to generating pull requests   
with updated dependencies.  
2. Dependency Management: Tools like Maven or Gradle are used to identify and manage dependencies, handling parent-child   
relationships and ensuring version compatibility.  
3. aPaaS Deployment: The deployment stage integrates with a cloud-based application platform, enabling rapid deployment   
and testing in a real-world environment.  
4. Security Integration: Prisma integrates with the pipeline to automatically scan for security vulnerabilities after the   
dependency upgrade and before deployment.  
5. Version Control (Git): Git is used to pull the source code, track changes in dependencies, and create pull requests for   
review and merging.  
  
Constraints and Challenges  
1. Dependency Conflicts: Upgrading multiple dependencies often leads to conflicts. It is crucial to handle transitive   
dependencies carefully to avoid breaking the application.  
2. Test Coverage: The success of the evergreening process depends heavily on the robustness of the test suite. Insufficient   
or missing tests can allow breaking changes to go unnoticed.  
3. Performance Impact: While upgrading dependencies can introduce performance improvements, it can also degrade performance   
in certain areas, requiring careful monitoring.  
4. Third-Party Library Compatibility: Some third-party libraries may not always be compatible with the latest versions of   
their dependencies, which can delay or block upgrades.  
5. Complex Dependency Trees: Applications with large and complex dependency graphs can take longer to analyze, upgrade,   
and test, which can impact pipeline performance and feedback speed.  
  
Conclusion  
Application evergreening, when executed via an automated Jenkins pipeline, can significantly improve the maintainability,   
security, and performance of a Java-based monolithic application. By keeping dependencies up to date, vulnerabilities are   
minimized, performance can be improved, and the application remains compatible with new versions of libraries. However,   
the success of such a system is reliant on comprehensive test coverage, careful handling of dependency conflicts, and   
continuous monitoring after deployment.  
  
This approach, while primarily focused on dependency resolution, also enhances security by integrating Prisma scanning and   
minimizes manual intervention through automation, making it an effective long-term strategy for large-scale Java   
applications.