Regulatory Package Management

Technical Overview

Applying proven software dependency management to regulatory compliance

3. Why Treating Regulatory Documents as Code Enables 30 Years of Learned Benefits

The Software Dependency Problem (Solved in 1990s-2000s)

Before package managers, software development faced identical problems:

Pre-Maven/NPM/pip era:

- Developers manually downloaded library JAR files from project websites
- No standard way to declare "I need library X version 2.3"
- Dependency conflicts: Library A needs XML parser v1.0, Library B needs v2.0
- No automated verification: "Did I get the correct file from a trusted source?"
- "Works on my machine" syndrome: Different developers had different library versions

Post-package manager:

- Declare dependencies in manifest file (pom.xml, package.json, requirements.txt)
- Package manager resolves transitive dependencies automatically
- Checksums verify integrity
- Version ranges allow compatible updates: "any 2.x version"
- Reproducible builds: Same inputs produce identical outputs

Direct Parallels to Regulatory Documents

Software Development	Regulatory Compliance
Library JAR file	Regulation PDF/XSD
Library version (e.g., 2.3.1)	Consolidated version date (e.g., 2024.01.09)
Dependency: "Needs library X"	Dependency: "RTS implements Article 9"
Transitive dependencies	Schema needs RTS needs base law
SHA-256 checksum	(Currently absent)
Maven Central repository	(Currently absent - our proposal)

pom.xml dependency declaration	(Currently manual)
mvn install downloads all files	(Currently manual downloads)
Version conflict detection	(Currently undetected)

Lessons from 30 Years of Package Management

Lesson 1: Namespacing Prevents Collisions

Structured coordinates like `eu.regulation.emir:law-648-2012:article-9` are unambiguous, unlike "Article 9" which could refer to EMIR, MiFIR, or SFTR.

Lesson 2: Semantic Versioning Communicates Intent

Hybrid approach combines date-based versioning for law (2024.01.09) with semantic versioning for technical artifacts (schemas-3.0-mar2023) to indicate breaking changes.

Lesson 3: Dependency Resolution Prevents Conflicts

Dependency graphs ensure RTS, schemas, and validation rules are compatible versions, preventing undefined behavior.

Lesson 4: Checksums Ensure Integrity

SHA-256 hashes verify document authenticity and detect corruption or tampering.

Lesson 5: Bill of Materials (BOM) Manages Complexity

One BOM request (`emir-bom:2024.11.0`) retrieves complete tested package: law + RTS + ITS + schemas + validation rules.

Lesson 6: Reproducibility Enables Auditability

pom.xml from Q2 2023 provides complete specification of exact regulatory versions used, enabling audit reproduction.

4. Maven Explainer

Apache Maven is a build automation and dependency management tool for Java projects, created in 2003. While initially designed for software builds, its dependency management concepts are applicable to any versioned artifacts.

Core Concepts

Coordinates (GAV)

Every artifact is uniquely identified by: `groupId : artifactId : version`

Example (regulations): `eu.regulation.emir : law-648-2012 : 2024.01.09`

POM (Project Object Model)

XML file describing artifact and its dependencies:

```
<project>
  <groupId>eu.regulation.emir</groupId>
  <artifactId>rts-2015-2205</artifactId>
  <version>2023.05.15</version>
  <dependencies>
    <dependency>
      <groupId>eu.regulation.emir</groupId>
      <artifactId>law-648-2012</artifactId>
      <version>2024.01.09</version>
    </dependency>
  </dependencies>
</project>
```

Dependency Resolution

When you declare a dependency, Maven:

- Downloads the POM file
- Reads its dependencies
- Recursively downloads transitive dependencies
- Detects version conflicts
- Verifies checksums
- Stores in local cache (~/.m2/repository/)

What Maven Provides

- Standardized coordinates: Globally unique artifact identification
- Transitive dependency resolution: Automatically fetch what you need
- Checksum verification: Integrity guarantees
- Version management: Track available versions, identify latest
- Local caching: Download once, reuse across projects
- Mature tooling: 20+ years of ecosystem development

What Maven Doesn't Provide

- Content analysis: Doesn't understand PDFs or extract obligations
- Semantic understanding: No knowledge of regulatory relationships
- Change tracking: Doesn't detect what changed between versions
- Workflow: No approval process, check-in/check-out, branching
- Search: Basic artifact search only, not full-text content search

5. Nexus Repository Manager Explainer

Sonatype Nexus Repository Manager is a repository server for storing and managing binary artifacts. It acts as a central hub for Maven artifacts (and other package formats: npm, PyPI, Docker, etc.).

Two editions: Nexus Repository OSS (free, open-source) and Nexus Repository Pro (commercial).

Core Functions

Artifact Storage: Stores artifacts in Maven format with automatic checksum generation (SHA-256, SHA-1, MD5) and metadata management.

Repository Types: Hosted (your artifacts), Proxy (cache from remote), Group (combine multiple repositories).

Access Control: Role-based access control (RBAC): anonymous read for public, authentication for premium, deploy permissions controlled.

REST API: Programmatic access for searching, uploading, and managing artifacts.

Web UI: Browser-based interface for browsing, searching, downloading, and viewing dependencies.

Infrastructure Requirements

Nexus Repository OSS:

- Cost: Free (open-source, Eclipse Public License)
- Server: 4-8 CPU cores, 8-16GB RAM recommended for production
- Storage: 1-10GB per regulatory regime (all versions)
- Deployment: Docker container, cloud-native (AWS, Azure, GCP)

Typical setup: AWS EC2 t3.large (~$60/month) + 1TB EBS storage (~$100/month) + data transfer = $200-500/month.

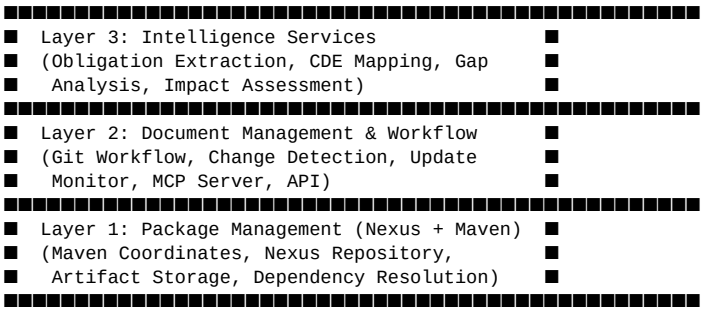
What Nexus Doesn't Provide

- Git workflow: No branching, pull requests, or code review
- Document processing: Doesn't parse PDFs or extract content
- Analysis: No understanding of regulatory semantics
- Notifications: Basic webhooks only, not "alert when EMIR updated"
- Gap analysis: No concept of entity profile or compliance requirements

6. NYQST Regulatory Package Manager: System Design

Architecture Overview

NYQST builds three layers on top of Nexus + Maven:



Layer 1: Package Management Foundation

Uses Maven + Nexus as-is, applying their patterns to regulatory documents.

Artifact types: Laws/Regulations (PDFs), Technical Standards (RTS/ITS), Schemas (XSD in ZIP), Validation Rules (Excel), BOMs (POM-only).

Repository structure:

- Public hosted repository: Freely available regulatory documents (anonymous read)
- Premium hosted repository: Analyzed documents (authenticated access, billing integration)

Client usage:

```
mvn dependency:get -Dartifact=eu.regulation:emir-bom:2024.11.0

# Downloads to ~/.m2/repository/eu/regulation/emir-bom/2024.11.0/
# Automatically fetches all dependencies listed in BOM
```

Layer 2: Document Management & Workflow

Git integration for document lifecycle:

- Check-in: Team member adds new regulatory document to Git
- Pull request: Changes reviewed by compliance team
- Approval: Legal sign-off required before merge
- CI/CD pipeline: Validates POM, computes checksums, runs quality checks, deploys to Nexus
- Git tag: Release tagged (e.g., emir-bom-2024.11.0)

Change detection: Monitor regulator websites (EUR-Lex, FCA), compare checksums, generate change reports, notify subscribers.

MCP Server: LLM agents can query regulatory repository, retrieve obligations, analyze changes between versions.

Layer 3: Intelligence Services

This layer is where NYQST adds commercial value beyond free document hosting.

Document Ingestion & Parsing: PDF text extraction (pdfplumber), OCR (Tesseract), structure identification (articles, sections), metadata enrichment (CELEX numbers, effective dates).

Obligation Extraction: Identify normative language ("shall", "must"), parse components (actor, action, object, recipient, timing, conditions), publish as Maven artifact with dependency on source document.

CDE Mapping: Map regulatory fields to CPMI-IOSCO Critical Data Elements, indicate harmonization across regimes (EMIR, MiFIR, SFTR, CFTC, etc.).

ISO 20022 Integration: Link regulatory fields to ISO 20022 message elements (message name, XPath, data type, occurrence).

Gap Analysis: Input entity profile (jurisdiction, activities, products), query canonical model, compute gaps (required - installed), prioritize (critical, recommended, optional).

Technology Stack

Core: Nexus OSS, Maven, Git, PostgreSQL

Analysis: Python, Neo4j (graph DB), PyPDF2/pdfplumber, spaCy/NLTK

Integration: FastAPI, MCP Server, Webhooks

Clients: Maven CLI, custom CLI (nyqst-reg), web dashboard

Access Tiers

Public (Free): Raw regulatory documents from public sources, basic Maven repository, version tracking, checksums.

Premium (Subscription): Public + structured obligations + CDE mappings + ISO 20022 links, API access, gap analysis, update notifications.

Enterprise (Custom): Premium + client-specific analysis + custom BOMs, dedicated support, custom integrations, priority updates.

Conclusion

The software development industry solved library dependency management 20+ years ago through package managers like Maven and repository servers like Nexus.

NYQST Regulatory Package Manager applies these proven patterns to regulatory compliance:

- **Maven** provides standardized coordinates, dependency resolution, and version management
- **Nexus** provides centralized storage, access control, and APIs
- **Git + workflows** add approval processes and change tracking
- **Analysis services** extract structured obligations, map to CDEs and ISO 20022, and enable gap analysis

The foundation (Layer 1) uses open-source tools with zero licensing cost. The value-add (Layer 3) is where commercial services differentiate. The result is a system that provides single-version-of-truth, automated dependency resolution, integrity verification, and auditability—benefits that the software industry has relied upon for decades, now applied to regulatory compliance.