Cross-Language Library Design

Lessons from Building JVM Data Connectors That Translate to Rust Crate Development

A journey through architectural patterns that transcend language boundaries. Insights from Capital One's Spark JMS connector project and their application to Rust ecosystem development.

By: **Venkata Surendra Reddy Appalapuram**

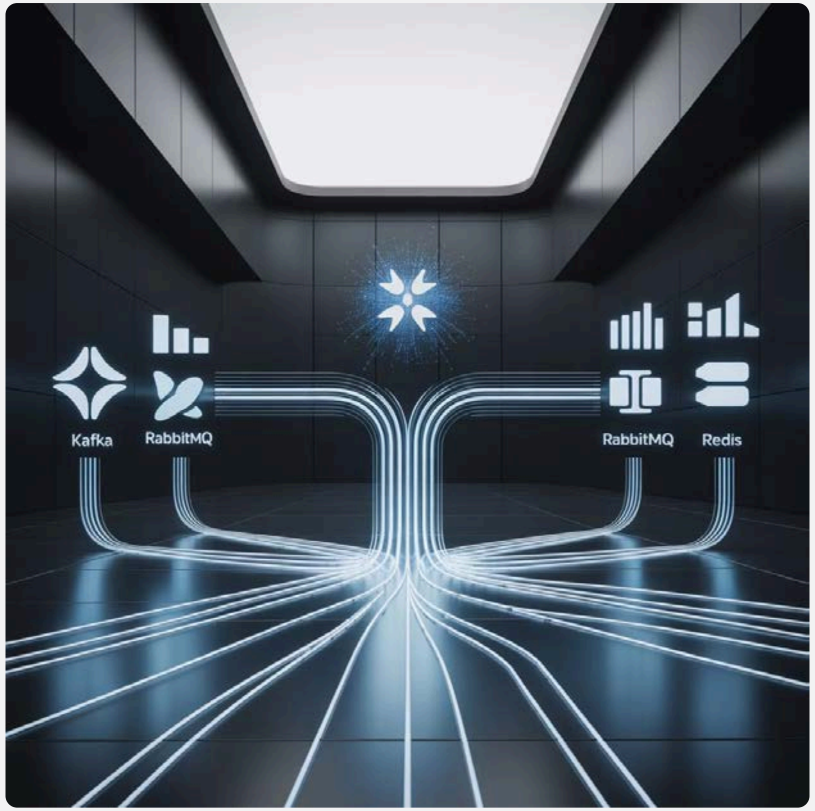
Agenda

Project Background

Capital One's Spark JMS Connector

- Enterprise-grade data connector for Apache Spark
- Enables stream processing from messaging systems
- Supports multiple JMS providers (ActiveMQ, IBM MQ, Solace)
- Critical for real-time data pipelines and fraud detection
- Performance-sensitive with strict reliability requirements

The library needed to handle diverse messaging patterns while maintaining consistent behavior and error handling across different broker implementations.



Universal Design Challenges

Library design faces similar fundamental challenges across language ecosystems

Abstraction Boundaries

Creating the right interfaces that hide implementation details while providing sufficient flexibility

Configuration Management

Balancing ease-of-use with the flexibility to customize behavior

Error Propagation

Communicating failures clearly while preserving context and recovery options

Extensibility

Enabling future enhancements without breaking backward compatibility

Performance Constraints

Minimizing overhead while maintaining safety and correctness guarantees

From JVM Interfaces to Rust Traits

The provider interface pattern we used in the JMS connector has a natural analog in Rust's trait system:

JVM Provider Interface

Java interfaces with dependency injection to swap implementations

Rust Trait System

Zero-cost abstractions with compile-time polymorphism

Separation of Concerns

How component isolation patterns transfer between ecosystems

JVM Implementation

- Message consumers isolated from connection management
- Acknowledgment strategies separated from message processing
- Error handlers decoupled from core business logic
- Configuration validation separate from usage

Used dependency injection and builder patterns to compose components

Rust Translation

- Module system naturally enforces boundaries
- Ownership model clarifies responsibility for resources
- Trait objects for runtime polymorphism when needed
- Type parameters for compile-time polymorphism

Rust's borrow checker enforces clean separation and prevents leaky abstractions

Error Handling Strategies

JVM Approach: Exceptions

- **Object-Oriented:** Relies on an inheritance hierarchy for runtime error representation.
- **Checked vs. Unchecked:** Enforces handling for API errors, while allowing programming errors to propagate.
- **Context & Recovery:** Emphasizes detailed messages, stack traces, and robust `catch` blocks for recovery.
- **Performance:** Consider implications of frequent exception throwing in high-performance contexts.

Rust Translation: Result Enum

- **Type-Safe:** `Result` explicitly encodes success or failure, forcing compile-time handling.
- **Custom Errors:** Uses custom error enums with `From` for cohesive, composable error types.
- **Contextual Chaining:** Libraries like `anyhow` and `thiserror` provide ergonomic context addition.
- **Pattern Matching:** Exhaustive `match` statements ensure all error cases are handled, preventing panics.

Comprehensive Testing Methodologies

Unit Testing Strategies

Testing individual components in isolation:

- JVM: Mockito for interface mocking
- Rust: Mock implementations of traits

Integration Testing

Testing across component boundaries:

- JVM: TestContainers for broker instances
- Rust: Similar container-based testing or feature flags

Property-Based Testing

Testing behavioral invariants:

- JVM: QuickTheories or jqwik
- Rust: proptest or quickcheck

Configuration Management Patterns

JVM Configuration Approach

- Builder pattern with sensible defaults
- Immutable configuration objects
- Validation at construction time
- Hierarchical configuration with overrides

Rust Translation

- Builder pattern with Default trait
- Type-safe configuration with compile-time validation
- const generics for static configuration
- Config structs with validation functions

Both approaches emphasize type safety and validation before use, but Rust can push more validation to compile time.

Performance Considerations

Different optimization approaches that achieve similar goals

10x	0	99.9%
Performance Improvement	Runtime Overhead	Reliability Target
Achieved in both ecosystems through careful design	Target for abstractions in performance-critical paths	Required for enterprise data processing systems

JVM Optimization Techniques

- Batch processing to amortize overhead
- Connection pooling and reuse
- Careful memory management to reduce GC pressure
- JIT-friendly code patterns

Rust Optimization Techniques

- Zero-cost abstractions via monomorphization
- Explicit memory management with lifetimes
- Compile-time evaluation when possible
- Fearless concurrency with ownership model

Documentation & API Design

- Consistent Naming Conventions

  Follow language idioms: camelCase for Java, snake_case for Rust. Establish clear, domain-specific terminology that remains consistent throughout the API.

- Progressive Disclosure

  Simple use cases should be simple to implement. Advanced features available but not required for basic usage. Both ecosystems benefit from tiered APIs with increasing complexity.

- Examples as Documentation

  Both Rustdoc and Javadoc support embedded examples. Our JVM connector provided example classes; Rust documentation can use doc tests that are automatically verified.

- Error Documentation

  Explicitly document all possible errors and recovery strategies. In Rust, this means documenting all Error variants that can be returned from each function.

Key Takeaways

1

Abstraction Principles Are Universal

Good interface design transcends language. Identify the right abstraction boundaries regardless of implementation language.

2

Leverage Language Strengths

Rust's ownership model and trait system provide compile-time guarantees that required runtime checks in JVM languages.

3

Testing Is Language-Agnostic

Comprehensive testing strategies translate well between ecosystems, though implementation details differ.

4

Document Intent, Not Just Implementation

Explaining why a design choice was made is as important as documenting how to use an API, regardless of language.

Thank You