Custom Extension and Agent Development Guide

Jan 2026

Table of Contents

Krista Extension Development Guide

Krista Custom Agent Development Guide

Krista Extension Development Guide

The Definitive Guide: A comprehensive resource for developing production-ready Krista extensions from initial architecture through deployment.

Part I: Foundation and Architecture

Introduction

Purpose and Scope

This comprehensive guide provides everything you need to develop robust, production-ready extensions for the Krista platform. It combines architectural guidance, implementation details, and best practices into a single authoritative resource that takes you from initial design through deployment and operations.

What You'll Learn

- **Architecture and Design**: - Connector framework principles and design patterns - Modular extension architecture with clear separation of concerns - Package structure and dependency management - Integration patterns for external systems
- **Implementation**: - Complete extension development lifecycle from setup to deployment - Java 21 development patterns and best practices - Advanced KSDK usage and dependency injection - Setup tab functionality for extension configuration - Event-driven programming and asynchronous processing
- **Quality and Operations**: - Comprehensive error handling and observability - Testing strategies for all extension components - Production deployment and monitoring - Troubleshooting common issues

Prerequisites

Required Knowledge: - Java 21 development environment - Gradle 8.4+ build system - Basic understanding of dependency injection (HK2) - Familiarity with JAX-RS for REST API development - Understanding of annotations and reflection

Key Concepts

- **Extension**: A plugin that extends Krista's functionality by integrating with external systems like Salesforce, ServiceNow, AWS, GitHub, and custom APIs.
- **Invoker**: A runtime instance of an extension with specific configuration. Multiple invokers can exist for the same extension with different settings.
- **Connector Framework**: An architectural pattern where extensions function as intelligent connectors that bridge the Krista platform and external systems.
- **Setup Tab**: The UI where users configure extension attributes, dependencies, authentication, and other settings before deployment.

- **Catalog Request**: Operations that interact with external systems (Change, Query, Wait for Event).
- **KSDK**: Krista Software Development Kit - provides core components and utilities for extension development.
- **Domain**: A logical grouping of related functionality within an extension, representing a specific area of integration.

How to Use This Guide

This guide is organized to support both linear reading and reference lookup:

- **For New Developers:** 1. Start with Part I (Foundation and Architecture) to understand core concepts 2. Work through Part II (Development Fundamentals) to set up your environment 3. Follow Part III (Core Implementation) to build your first extension 4. Implement Part IV (Setup Tab) to make your extension configurable 5. Explore Part V (Advanced Features) as needed for your use case 6. Apply Part VI (Quality and Operations) before production deployment
- **For Experienced Developers:** - Use the table of contents to jump to specific topics - Reference Part VII for complete implementation patterns - Consult Part VI for production best practices - Use Part IV for Setup tab implementation details
- **For Troubleshooting:** - Check Section 29 (Troubleshooting) for common issues - Review Section 26 (Error Handling) for debugging strategies - Consult Section 25 (Testing Strategy) for validation approaches

---

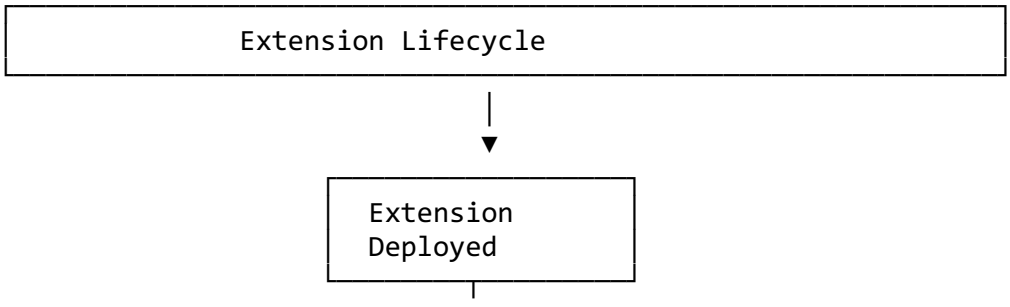
Getting Started

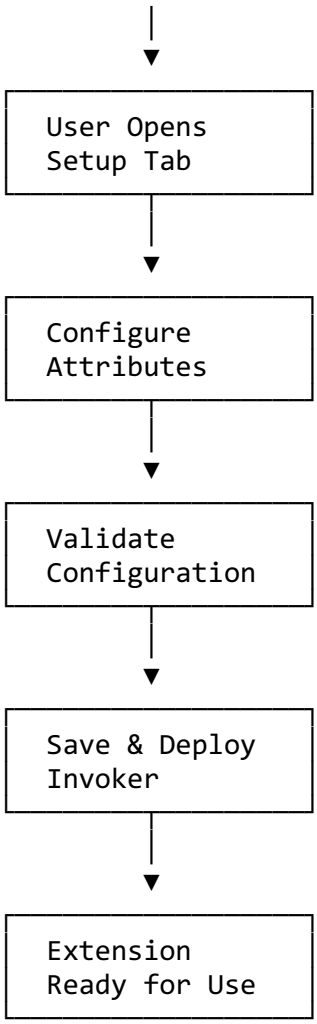
Extension Architecture Overview

Krista extensions follow a modular architecture with these key components:

- **Extension Class**: Main entry point with lifecycle management and metadata
- **Catalog Requests**: System integration operations (Change, Query, Wait for Event)
- **Entity Definitions**: Data model representations for external system concepts
- **Invoker Requests**: Event handlers and interface operations
- **Service Integration**: External API communication patterns
- **Setup Tab**: Configuration interface for extension attributes and dependencies

Extension Lifecycle





Development Environment Setup

1. Create Extension Project

```
mkdir my-extension
cd my-extension
gradle init --type java-library
```

2. Configure Build Dependencies

```
// build.gradle
plugins {
    id 'java-library'
    id 'maven-publish'
}

java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(21))
    }
```

```
}

repositories {
    maven {
        url = "<valid maven url>"
    }
    mavenCentral()
    mavenLocal()
}

dependencies {
    // Core Krista Extension APIs (from krista-service-apis-java repository)
    annotationProcessor 'app.krista:extension-impl-anno-processors:3.5.0'
    implementation 'app.krista:extension-api:3.5.0'
    implementation 'org.glassfish.hk2:hk2-api:3.0.3'

    // JAX-RS and HTTP Client
    implementation 'org.glassfish.jersey.core:jersey-client:3.1.0'
    implementation 'org.glassfish.jersey.media:jersey-media-json-jackson:3.1.
0'

    // Testing
    testImplementation 'junit:junit:4.13.2'
    testImplementation 'org.mockito:mockito-core:4.11.0'
}
```

3. Create Basic Extension Structure

```
my-extension/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/yourcompany/krista/extension/myextension/
│   │   │       ├── MyExtension.java
│   │   │       ├── integration/          # External system integration
│   │   │       ├── transformation/       # Data transformation
│   │   │       ├── controller/           # Catalog request handlers
│   │   │       ├── entity/               # Krista entity definitions
│   │   │       └── config/               # Configuration management
│   │   └── resources/
│   │       ├── META-INF/
│   │       │   └── logo.png
│   │       └── docs/
│   └── test/
│       └── java/
├── build.gradle
└── settings.gradle
```

Quick Start: Your First Extension

Let's create a minimal working extension:

```
package com.yourcompany.krista.extension.myextension;

import app.krista.extension.impl.anno.Extension;
import app.krista.extension.impl.anno.Domain;
import app.krista.extension.impl.anno.CatalogRequest;
import app.krista.extension.impl.anno.Field;
import app.krista.extension.request.ExecutionMode;
import app.krista.extension.response.ExtensionResponse;

@Java(version = Java.Version.JAVA_21)
@Extension(
    version = "1.0.0",
    name = "My First Extension",
    description = "A simple extension to get started",
    logo = "/META-INF/logo.png",
    implementingDomainIds = {"my-domain"},
    supportingModes = {ExecutionMode.TEST}
)
@Domain(
    id = "catEntryDomain_my-domain-12345",
    name = "My Domain",
    ecosystemId = "catEntryEcosystem_custom",
    ecosystemName = "Custom Integrations",
    ecosystemVersion = "v1.0.0"
)
public class MyExtension {

    @CatalogRequest(
        id = "localDomainRequest_hello-world",
        name = "Say Hello",
        description = "Returns a greeting message",
        area = "Greetings",
        type = CatalogRequest.Type.QUERY_SYSTEM
    )
    @Field.Text(name = "Name", required = true)
    public ExtensionResponse sayHello(
        @Field.Text(name = "Name") String name
    ) {
        String greeting = "Hello, " + name + "!";
        return ExtensionResponse.success()
            .withData("greeting", greeting);
    }
}
```

Building and Testing

Build the extension:

```
gradle clean build
```

Run tests:

```
gradle test
```

Package for deployment:

```
gradle jar
```

The compiled extension JAR will be in *`build/libs/`*.

Next Steps

Now that you have a basic extension running, you'll learn:

1. **Architecture principles** - How to design scalable, maintainable extensions
2. **Module structure** - Organizing code into logical layers
3. **Setup tab implementation** - Making extensions configurable
4. **Advanced features** - Events, storage, custom tabs, and more

---

Connector Framework Architecture

Design Principle

The Connector Framework represents a fundamental shift in how Krista extensions are architected. Rather than building monolithic applications that attempt to handle multiple concerns, extensions function as **intelligent connectors** that serve as specialized bridges between the Krista platform and external systems. This architectural pattern recognizes that modern enterprise software environments are inherently distributed and require sophisticated integration capabilities.

Each extension should maintain a laser focus on integrating with one external system or a closely related group of systems (such as the Microsoft 365 suite or Google Workspace). This focused approach ensures that extensions remain maintainable, testable, and performant while providing deep, specialized integration capabilities.

The connector pattern addresses several critical challenges in enterprise integration:

System Complexity Management: External systems often have complex APIs, authentication mechanisms, and data models. The connector framework encapsulates this complexity, presenting a clean, Krista-native interface to end users while handling all the intricate details of external system communication behind the scenes.

Protocol Diversity: Different external systems use various communication protocols (REST, GraphQL, SOAP, WebSockets, message queues). The connector framework abstracts these protocol differences, allowing Krista to interact with all external systems through a consistent interface pattern.

Evolution and Change Management: External systems frequently update their APIs, change authentication methods, or modify data structures. By isolating external system interactions within dedicated connector extensions, these changes can be managed without impacting the broader Krista platform or other extensions.

Core Responsibilities

API Integration

The primary responsibility of any connector extension is to handle all communication with third-party systems. This includes managing HTTP requests, handling authentication flows, processing responses, and dealing with API-specific quirks or limitations. The extension must understand the external system's API patterns, rate limiting requirements, pagination mechanisms, and error response formats.

For example, when integrating with Microsoft Graph API, the extension must handle OAuth 2.0 authentication flows, manage access token refresh cycles, understand the specific pagination patterns used by different Graph endpoints, and properly handle the various error response formats that Microsoft returns for different failure scenarios.

Protocol Abstraction

External systems expose their functionality through various protocols and patterns. A well-designed connector extension hides these protocol complexities from the Krista platform, presenting a unified interface regardless of whether the external system uses REST APIs, GraphQL endpoints, SOAP services, or proprietary protocols.

This abstraction layer is crucial for maintaining consistency across different integrations. Users interacting with a Salesforce connector should have a similar experience to those using a ServiceNow connector, even though these systems have vastly different underlying APIs and data models.

State Management

Connector extensions must maintain various types of state information including authentication tokens, session identifiers, connection pools, and cached data. This state management must be robust, secure, and efficient. The extension needs to handle token expiration and refresh, connection failures and recovery, and cache invalidation scenarios.

State management becomes particularly complex in multi-tenant environments where different invokers may have different credentials, permissions, and configuration settings. The extension must isolate state appropriately while maintaining performance and security.

Error Translation

External systems generate errors in their own formats and terminology. A connector extension must translate these external errors into meaningful, actionable messages that Krista users can understand and act upon. This includes mapping technical error codes to user-friendly messages, providing context about what operation failed and why, and suggesting remediation steps when possible.

Architectural Benefits

Modularity

The connector framework enables true modularity in extension architecture. Each connector can be developed, tested, deployed, and scaled independently. This modularity provides several advantages:

Independent Development Cycles: Teams can work on different connectors simultaneously without coordination overhead. A team working on a Slack integration doesn't need to coordinate with a team building a Jira connector.

Selective Deployment: Organizations can choose which connectors to deploy based on their specific needs, reducing resource consumption and attack surface.

Isolated Failure Domains: If one connector experiences issues, it doesn't impact other connectors or the core Krista platform.

Reusability

Well-designed connectors can be shared across different Krista workspaces, organizations, and use cases. A Microsoft 365 connector built for email management can be reused for calendar integration, document management, or user provisioning scenarios. This reusability reduces development effort and ensures consistent behavior across different implementations.

Maintainability

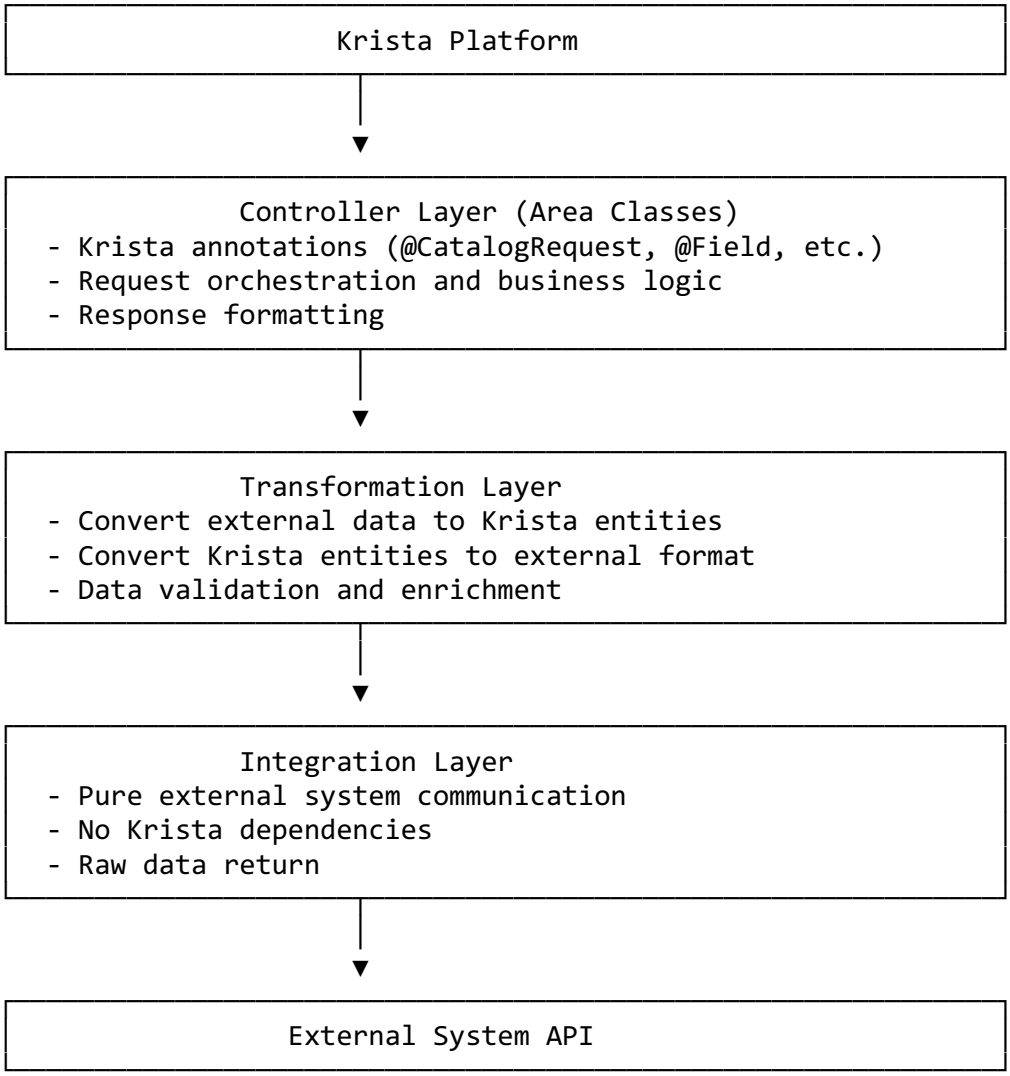
By isolating external system interactions within dedicated connectors, maintenance becomes more manageable. When Microsoft updates their Graph API, only the Microsoft connector needs to be updated. When a new version of Salesforce is released, only the Salesforce connector requires attention. This isolation dramatically reduces the maintenance burden and risk associated with external system changes.

Testability

The clear separation of concerns inherent in the connector framework enables comprehensive testing strategies. Each layer of the connector can be tested independently, and the isolation of external system interactions makes it possible to create reliable, repeatable test suites that can run both with mocked external systems and against real external APIs.

Layered Architecture Pattern

Extensions follow a strict layered architecture:



Data Flow:
1. Krista platform sends request to Controller Layer
2. Controller validates input and applies business rules
3. Controller calls Transformation Layer with validated parameters
4. Transformation Layer calls Integration Layer to fetch raw data
5. Integration Layer communicates with external system
6. Raw data flows back up through Transformation Layer (converted to Krista entities)
7. Controller formats response and returns to Krista platform

Key Principle: Each layer depends only on layers below it. The Integration Layer has zero Krista dependencies and can be used in any Java application.

Extension Sub-modules and Package Structure

The modular architecture of Krista extensions is built around five core sub-modules, each with distinct responsibilities and clear boundaries. This modular approach ensures separation of concerns, enables independent testing, and facilitates maintenance and evolution of the extension over time.

Required Sub-modules

1. External Integration Module

Purpose: Pure third-party system communication
Location: *`com.yourcompany.integration.{system}/`*

The External Integration Module serves as the foundation of the connector architecture. This module is responsible for all direct communication with external systems and must be completely isolated from Krista-specific code.

Key Responsibilities:

- **HTTP Client Management**: Configure and manage HTTP clients with appropriate timeouts, connection pooling, and retry policies
- **Authentication Handling**: Implement OAuth flows, API key management, token refresh cycles, and credential storage
- **Request/Response Processing**: Handle request serialization, response deserialization, and protocol-specific formatting
- **Error Management**: Catch and translate external system errors into meaningful exceptions
- **Rate Limiting**: Implement client-side rate limiting to respect external system constraints
- **Connection Health**: Monitor connection health and implement circuit breaker patterns for resilience

Design Principles:

- **Zero Krista Dependencies**: This module must not import any Krista-specific classes or interfaces
- **Stateless Operations**: Each method should be independent and not rely on instance state
- **Raw Data Focus**: Return data exactly as received from external systems without interpretation
- **Comprehensive Logging**: Log all external interactions for debugging and monitoring

Common Pitfalls to Avoid: - Don't include business logic or data transformation in this module - Avoid hardcoding configuration values - use dependency injection - Don't catch and suppress exceptions - let them bubble up with context - Avoid mixing different external systems in the same integration module

2. Data Transformation Module

Purpose: Convert between external and Krista data formats
Location: *`com.yourcompany.krista.extension.{system}.transformation/`*

The Data Transformation Module acts as a translation layer between the raw data structures returned by external systems and the clean, Krista-native entities that the platform expects.

Key Responsibilities: -

- **Bidirectional Mapping**: Transform data from external format to Krista format and vice versa
- **Data Validation**: Ensure transformed data meets Krista entity requirements and business rules
- **Field Mapping**: Handle complex field mappings including nested objects, arrays, and computed fields
- **Data Enrichment**: Add derived fields, calculate values, and enhance data with additional context
- **Format Conversion**: Convert between different data types, date formats, and encoding schemes
- **Default Value Management**: Provide sensible defaults for missing or null external data

Transformation Patterns: - *Direct Field Mapping*: External fields map directly to Krista fields with minimal processing - *Data Aggregation*: Combining multiple external fields into a single Krista field - *Data Derivation*: Computing new fields based on external data - *Data Filtering*: Selectively including or excluding data based on business rules

Error Handling Strategy: - Validate all input data before transformation - Provide meaningful error messages that identify the specific field or transformation that failed - Implement fallback strategies for non-critical data transformations - Log transformation failures with sufficient context for debugging

3. Controller Module (Area Classes)

Purpose: Krista platform request handlers and orchestration
Location: *`com.yourcompany.krista.extension.{system}.controller/`*

The Controller Module serves as the orchestration layer that coordinates between the integration and transformation modules while implementing the Krista platform interfaces.

This is where Krista-specific annotations are applied and where the extension's public API is defined.

Key Responsibilities:

- **Request Orchestration**: Coordinate calls between integration and transformation layers
- **Krista Platform Integration**: Implement Krista annotations and interfaces correctly
- **Business Logic Application**: Apply extension-specific business rules and workflows
- **Response Formatting**: Format responses according to Krista platform expectations
- **Error Handling**: Convert technical exceptions into user-friendly error responses
- **Security Enforcement**: Apply authorization checks and data access controls
- **Performance Optimization**: Implement caching, batching, and other performance strategies

Design Patterns: - Use dependency injection for all service dependencies - Implement comprehensive input validation before processing - Apply the single responsibility principle - one Area class per functional domain - Use consistent error handling and response formatting patterns

4. Entity Module

Purpose: Krista entity definitions with annotations
Location: *`com.yourcompany.krista.extension.{system}.entity/`*

The Entity Module defines the data structures that represent external system concepts within the Krista platform. These entities serve as the contract between the extension and the Krista platform, defining what data is available and how it should be presented to users.

Key Responsibilities: Apply appropriate data classification and access control annotations

Best Practices: - Use Java records for immutable entity definitions when appropriate - Apply comprehensive field annotations to control UI rendering - Include appropriate search and classification annotations - Document entity relationships and dependencies clearly

5. Configuration Module

Purpose: Extension settings and invoker attributes
Location: *`com.yourcompany.krista.extension.{system}.config/`*

The Configuration Module manages all extension settings, connection parameters, and user preferences. This module is critical for making extensions flexible and deployable across different environments and use cases.

Key Responsibilities:

- **Connection Settings**: Define and validate external system connection parameters
- **Feature Flags**: Implement toggles for optional functionality
- **Environment Configuration**: Handle differences between development, staging, and production environments
- **User Preferences**: Manage user-specific settings and customizations
- **Security Configuration**: Handle secure storage and retrieval of sensitive configuration data
- **Validation Logic**: Implement comprehensive validation for all configuration parameters

Module Dependencies and Data Flow

The dependency structure follows a strict layered architecture pattern:

```
Controller Layer
    ↓ (depends on)
Transformation Layer
    ↓ (depends on)
Integration Layer
    ↓ (depends on)
External System APIs
```

Data Flow Pattern:

1. **Inbound Request**: Krista platform sends request to Controller Layer
2. **Parameter Validation**: Controller validates input parameters and applies business rules
3. **Integration Call**: Controller calls Transformation Layer with validated parameters
4. **External System Call**: Transformation Layer calls Integration Layer to fetch raw data
5. **Data Transformation**: Transformation Layer converts raw data to Krista entities
6. **Response Formatting**: Controller formats transformed data into Krista response format
7. **Response Return**: Formatted response is returned to Krista platform

Dependency Injection Pattern: Each layer should use dependency injection to access services from lower layers. This enables proper testing, configuration management, and loose coupling between components.

Error Propagation: Errors should propagate up through the layers, with each layer adding appropriate context and handling layer-specific concerns. The Controller Layer is responsible for converting technical exceptions into user-friendly error messages.

Complete Package Structure

```
com.yourcompany.krista.extension.{system}/
├── {System}Extension.java                    # Main extension class
├── integration/                               # Third-party system package
│   ├── client/
│   │   ├── {System}ApiClient.java
│   │   ├── {System}AuthClient.java
│   │   └── {System}WebhookClient.java
│   ├── model/
│   │   ├── {System}User.java
│   │   ├── {System}Document.java
│   │   └── {System}Response.java
│   ├── auth/
│   │   ├── TokenManager.java
│   │   ├── AuthenticationProvider.java
│   │   └── CredentialStore.java
│   ├── exception/
│   │   ├── {System}ApiException.java
│   │   ├── AuthenticationException.java
│   │   └── RateLimitException.java
│   └── util/
│       ├── HttpClientFactory.java
│       ├── RequestBuilder.java
│       └── ResponseParser.java
├── transformation/                            # Data transformation layer
│   ├── UserTransformer.java
│   ├── DocumentTransformer.java
│   └── TransformationUtils.java
├── controller/                                # Krista controller layer
│   ├── {System}UserArea.java
│   ├── {System}DocumentArea.java
│   └── {System}WebhookArea.java
├── entity/                                    # Krista entity definitions
│   ├── KristaUser.java
│   ├── KristaDocument.java
│   └── KristaMetadata.java
├── config/                                    # Configuration management
│   ├── {System}ExtensionConfig.java
│   ├── ConnectionSettings.java
│   └── FeatureFlags.java
└── util/                                      # Shared utilities
    ├── Constants.java
    ├── ValidationUtils.java
    └── LoggingUtils.java
```

Part II: Development Fundamentals

Project Structure and Build Configuration

Gradle Build Configuration

build.gradle:

```
plugins {
    id 'java-library'
    id 'maven-publish'
}

java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(21))
    }
}

repositories {
    mavenCentral()
    maven {
        url "https://krista.repository.url"  // Your Krista repository
    }
}

dependencies {
    // Core Krista Extension APIs (from krista-service-apis-java repository)
    annotationProcessor 'app.krista:extension-impl-anno-processors:3.5.0'
    implementation 'app.krista:extension-api:3.5.0'
    implementation 'app.krista:ksdk-api:3.5.0'
    implementation 'app.krista:model-api:3.5.0'

    // JAX-RS and HTTP Client
    implementation 'org.glassfish.jersey.core:jersey-client:3.1.0'
    implementation 'org.glassfish.jersey.media:jersey-media-json-jackson:3.1.
0'
    implementation 'org.glassfish.jersey.inject:jersey-hk2:3.1.0'

    // JSON Processing
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.15.2'
    implementation 'com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.
15.2'

    // Logging
    implementation 'org.slf4j:slf4j-api:2.0.7'
    implementation 'ch.qos.logback:logback-classic:1.4.8'

    // Testing
```

```
    testImplementation 'junit:junit:4.13.2'
    testImplementation 'org.mockito:mockito-core:4.11.0'
    testImplementation 'org.assertj:assertj-core:3.24.2'
}

tasks.withType(JavaCompile) {
    options.encoding = 'UTF-8'
    options.compilerArgs += ['-parameters']  // Preserve parameter names
}

test {
    useJUnit()
    testLogging {
        events "passed", "skipped", "failed"
        exceptionFormat "full"
    }
}
```

Project Directory Structure

```
my-extension/
├── build.gradle
├── settings.gradle
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/yourcompany/krista/extension/myextension/
│   │   │       ├── MyExtension.java                # Main extension class
│   │   │       ├── integration/                    # External system integr
ation
│   │   │       │   ├── client/
│   │   │       │   │   ├── ApiClient.java
│   │   │       │   │   └── AuthClient.java
│   │   │       │   ├── model/
│   │   │       │   │   ├── ExternalUser.java
│   │   │       │   │   └── ExternalResponse.java
│   │   │       │   ├── auth/
│   │   │       │   │   └── TokenManager.java
│   │   │       │   └── exception/
│   │   │       │       └── ApiException.java
│   │   │       ├── transformation/                 # Data transformation
│   │   │       │   ├── UserTransformer.java
│   │   │       │   └── TransformationUtils.java
│   │   │       ├── controller/                     # Catalog request handle
rs
│   │   │       │   ├── UserArea.java
│   │   │       │   └── DocumentArea.java
│   │   │       ├── entity/                         # Krista entities
│   │   │       │   ├── KristaUser.java
│   │   │       │   └── KristaDocument.java
```

```
│   │   │   ├── config/                          # Configuration
│   │   │   │   └── ExtensionConfig.java
│   │   │   └── util/                            # Utilities
│   │   │       └── Constants.java
│   │   └── resources/
│   │       ├── META-INF/
│   │       │   ├── logo.png
│   │       │   └── MANIFEST.MF
│   │       ├── docs/                            # Documentation
│   │       │   └── index.html
│   │       └── application.properties
│   └── test/
│       └── java/
│           └── com/yourcompany/krista/extension/myextension/
│               ├── integration/
│               │   └── ApiClientTest.java
│               ├── transformation/
│               │   └── UserTransformerTest.java
│               └── controller/
│                   └── UserAreaTest.java
└── README.md
```

Extension Development with Java 21

Java 21 Features for Extensions

Extensions can utilize Java 21's advanced features for cleaner, more maintainable code:

Pattern Matching in Switch Expressions:

```
public String processData(Object data) {
    return switch (data) {
        case CustomerData(var name, var email, var created) ->
            "Customer: %s (%s) created at %s".formatted(name, email, created)
;
        case String s -> "String data: " + s;
        case null -> "No data provided";
        default -> "Unknown data type";
    };
}
```

Record Classes for Immutable Data:

```
public record CustomerData(String name, String email, LocalDateTime created)
{
    // Compact constructor for validation
    public CustomerData {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("Name cannot be blank");
```

```
        }
    }
}
```

Text Blocks for Multi-line Strings:

```
private static final String SQL_QUERY = """
    SELECT u.id, u.name, u.email, u.created_at
    FROM users u
    WHERE u.status = 'active'
      AND u.created_at > ?
    ORDER BY u.created_at DESC
    """;
```

Virtual Threads for High-Concurrency:

```
public List<Result> fetchDataConcurrently(List<String> ids) {
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        List<Future<Result>> futures = ids.stream()
            .map(id -> executor.submit(() -> fetchData(id)))
            .toList();

        return futures.stream()
            .map(future -> {
                try {
                    return future.get();
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
            })
            .toList();
    }
}
```

Sealed Classes for Controlled Hierarchies:

```
public sealed interface ExtensionResult permits Success, Failure, Pending {
    record Success(Object data) implements ExtensionResult {}
    record Failure(String error) implements ExtensionResult {}
    record Pending(String message) implements ExtensionResult {}
}
```

Java 21 Best Practices for Extensions

- **Pattern matching in switch expressions**: Simplify complex conditional logic
- **Record classes for immutable data**: Perfect for DTOs and entity representations
- **Text blocks for multi-line strings**: Ideal for SQL queries, JSON templates, and documentation
- **Virtual threads for high-concurrency scenarios**: Excellent for handling multiple external API calls

- **Sealed classes**: Provide controlled inheritance hierarchies for extension types

---

Extension Annotations Reference

All extension annotations are available in the *krista-service-apis-java* repository under the *extension/impl/anno/* directory. These annotations provide a declarative way to define extension metadata, request handlers, field definitions, and deployment configurations.

Core Annotations

@Extension

Package: *app.krista.extension.impl.anno.Extension*
Purpose: Marks the main extension class and defines core extension metadata.

Key Parameters: -
version(): Extension version (default: system property)
name(): Display name for the extension
description(): Extension description
logo(): Path to extension logo (default: "/META-INF/logo.png")
implementingDomainIds(): Array of domain IDs this extension implements
jaxrsId(): JAX-RS configuration identifier (default: "rest")
supportsInvokerProvisioning(): Enable invoker provisioning (default: false)
requireWorkspaceAdminRights(): Require admin rights (default: false)
supportingModes(): Array of supported execution modes
implementationModel(): Implementation model version (default: Krista_3_O)

Example:

```
import app.krista.extension.impl.anno.Extension;
import app.krista.extension.request.ExecutionMode;

@Extension(
    version = "2.1.0",
    name = "GitHub Integration",
    description = "Comprehensive GitHub API integration for repository manage
ment",
    logo = "/META-INF/github-logo.png",
    implementingDomainIds = {"github", "version-control"},
    jaxrsId = "rest",
    supportsInvokerProvisioning = true,
    requireWorkspaceAdminRights = false,
    supportingModes = {ExecutionMode.TEST, ExecutionMode.ACTIVE_MQ},
    implementationModel = Extension.ImplementationModel.Krista_4_O
)
public class GitHubExtension {
    // Extension implementation
}
```

@Domain

Package: *app.krista.extension.impl.anno.Domain*
Purpose: Defines domain metadata for the extension. Can be repeated using @Domains.

Parameters: -
id(): Unique domain identifier (required)
name(): Domain display name (required)
ecosystemId(): Ecosystem identifier (required)
ecosystemName(): Ecosystem display name (required)
ecosystemVersion(): Ecosystem version (required)

Example:

```
@Domain(
    id = "catEntryDomain_github-12345",
    name = "GitHub",
    ecosystemId = "catEntryEcosystem_development-tools",
    ecosystemName = "Development Tools",
    ecosystemVersion = "v2.1.0"
)
public class GitHubExtension {
    // Extension implementation
}
```

Request Annotations

@CatalogRequest

Package: *app.krista.extension.impl.anno.CatalogRequest*
Purpose: Defines catalog request operations that interact with external systems.

Parameters: -
id(): Unique request identifier (required)
name(): Request display name (required)
area(): Functional area for grouping (required)
type(): Request type (required)
description(): Request description (required)

Request Types: - *CHANGE_SYSTEM*: Modifies external system state - *QUERY_SYSTEM*: Retrieves data without modification - *WAIT_FOR_EVENT*: Waits for external events

Example:

```
@CatalogRequest(
    id = "localDomainRequest_create-repository",
    name = "Create Repository",
    description = "Create a new GitHub repository with specified configuratio
n",
    area = "Repository Management",
```

```
    type = CatalogRequest.Type.CHANGE_SYSTEM
)
@Field.Text(name = "Repository Name", required = true)
@Field.Boolean(name = "Private Repository", required = false)
public ExtensionResponse createRepository(
    @Field.Text(name = "Repository Name") String repositoryName,
    @Field.Boolean(name = "Private Repository") boolean isPrivate) {
    // Implementation
    return ExtensionResponse.success();
}
```

@InvokerRequest

Package: *app.krista.extension.impl.anno.InvokerRequest*
Purpose: Defines invoker lifecycle and event handling operations.

Parameters: - *value()*: InvokerRequest.Type enum value (required)

Complete Invoker Request Types:

Type	Purpose	Method Signature	When Called
AUTHENTICATOR	Provides authentication handler	*RequestAuthenticator method()*	When authentication is needed
CUSTOM_TABS	Defines custom UI tabs	*Map<String, Object> method()*	When rendering extension UI
GET_DEFINITIONS	Returns entity definitions	*List<EntityDefinition> method()*	When loading entity metadata
INVOKER_INSTALLED	Called on first installation	*void method()*	Once when invoker is first installed
INVOKER_LOADED	Called when invoker loads	*void method()*	Every time invoker loads/reloads
INVOKER_UPDATED	Called on configuration update	*void method(Map<String, Object> oldAttrs, Map<String, Object> newAttrs)*	When attributes change
INVOKER_REMOVED	Called before removal	*void method()*	Before invoker is deleted
INVOKER_UNLOADED	Called when invoker unloads	*void method()*	When invoker is unloaded
PROVISION	Handles entity provisioning	*void method(EntityDefinition definition)*	When provisioning entities
TEST_CONNECTION	Validates connection	*void method()*	When user tests connection

Type	Purpose	Method Signature	When Called
VALIDATE_ATTRIBUTES	Validates configuration	*void method(Map<String, Object> attributes)*	Before saving configuration
EVENT_DELIVERED	Handles delivered events	*void method(EventSubscription subscription, Map<String, Object> eventData)*	When event is delivered
RUNTIME_CATALOG_REQUEST	Runtime catalog execution	*Map<String, Object> method(CatalogRequest request, Map<String, Object> params)*	Dynamic catalog request execution
RUNTIME_API_REQUEST	Runtime API execution	*ProtoResponse method(String pathElement, ProtoRequest request)*	Dynamic API request execution
REGISTER_EVENT_LISTENER	Register event listener	*void method(WaitForEventListener listener)*	When registering for events
UNREGISTER_EVENT_LISTENER	Unregister event listener	*void method(WaitForEventListener listener)*	When unregistering from events
CATALOG_REQUEST_PROVISION	Provision catalog request	*void method(CatalogRequest request, Object data)*	When provisioning catalog requests
GET_CATALOG_REQUESTS	Get available catalog requests	*Map<String, Object> method()*	When loading catalog request metadata
PREPARE_CHANGE_ROUTING_ID	Prepare change routing	*void method(String routingId)*	Before routing change requests

Lifecycle Examples:

```
@InvokerRequest(InvokerRequest.Type.INVOKER_INSTALLED)
public void onInvokerInstalled() {
    logger.info("Extension invoker installed for the first time");
    // One-time initialization: create database tables, set up webhooks, etc.
    initializeDatabase();
    registerWebhooks();
}

@InvokerRequest(InvokerRequest.Type.INVOKER_LOADED)
public void onInvokerLoaded() {
    logger.info("Extension invoker loaded successfully");
    // Initialize resources: connections, caches, thread pools
```

```
    initializeResources();
}

@InvokerRequest(InvokerRequest.Type.INVOKER_UPDATED)
public void onInvokerUpdated(Map<String, Object> oldAttributes, Map<String, O
bject> newAttributes) {
    logger.info("Invoker configuration updated");

    // Compare old and new attributes
    String oldApiUrl = (String) oldAttributes.get("API URL");
    String newApiUrl = (String) newAttributes.get("API URL");

    if (!Objects.equals(oldApiUrl, newApiUrl)) {
        logger.info("API URL changed from {} to {}", oldApiUrl, newApiUrl);
        // Reinitialize connection with new URL
        reinitializeConnection(newApiUrl);
    }
}

@InvokerRequest(InvokerRequest.Type.INVOKER_UNLOADED)
public void onInvokerUnloaded() {
    logger.info("Extension invoker unloading");
    // Clean up resources: close connections, clear caches
    cleanupResources();
}

@InvokerRequest(InvokerRequest.Type.INVOKER_REMOVED)
public void onInvokerRemoved() {
    logger.info("Extension invoker being removed");
    // Final cleanup: delete data, unregister webhooks
    deleteExtensionData();
    unregisterWebhooks();
}

@InvokerRequest(InvokerRequest.Type.TEST_CONNECTION)
public void testConnection() {
    // Validate connection settings
    try {
        apiClient.ping();
        logger.info("Connection test successful");
    } catch (Exception e) {
        logger.error("Connection test failed", e);
        throw new RuntimeException("Connection test failed: " + e.getMessage(
));
    }
}

@InvokerRequest(InvokerRequest.Type.VALIDATE_ATTRIBUTES)
public void validateAttributes(Map<String, Object> attributes) {
    List<String> errors = new ArrayList<>();
```

```
    // Validation logic
    String apiKey = (String) attributes.get("API Key");
    if (apiKey == null || apiKey.length() < 32) {
        errors.add("API Key must be at least 32 characters");
    }

    if (!errors.isEmpty()) {
        throw new ValidationException(String.join(", ", errors));
    }
}

@InvokerRequest(InvokerRequest.Type.EVENT_DELIVERED)
public void handleEvent(EventSubscription subscription, Map<String, Object> e
ventData) {
    String eventType = (String) eventData.get("eventType");
    logger.info("Received event: {}", eventType);

    // Process event based on type
    switch (eventType) {
        case "user.created" -> handleUserCreated(eventData);
        case "user.updated" -> handleUserUpdated(eventData);
        case "user.deleted" -> handleUserDeleted(eventData);
        default -> logger.warn("Unknown event type: {}", eventType);
    }
}

@InvokerRequest(InvokerRequest.Type.PROVISION)
public void provision(EntityDefinition definition) {
    logger.info("Provisioning entity: {}", definition.getName());
    // Create entity in external system
    createEntityInExternalSystem(definition);
}
```

@SubCatalogRequest

Package: *app.krista.extension.impl.anno.SubCatalogRequest*
Purpose: Defines sub-catalog request operations for nested workflows and helper functions.

Parameters: -
name(): Request display name (required)
type(): Request type from CatalogRequest.Type (required)
description(): Request description (required)

Example:

```
import app.krista.extension.impl.anno.SubCatalogRequest;
import app.krista.extension.impl.anno.CatalogRequest;
```

```
@SubCatalogRequest(
    name = "Validate Repository Settings",
    type = CatalogRequest.Type.QUERY_SYSTEM,
    description = "Validate repository configuration before creation"
)
public boolean validateRepositorySettings(String repositoryName, boolean isPr
ivate) {
    // Validation logic
    if (repositoryName == null || repositoryName.isEmpty()) {
        return false;
    }
    // Check if repository name is valid
    return repositoryName.matches("^[a-zA-Z0-9_-]+$");
}

@SubCatalogRequest(
    name = "Calculate Repository Size",
    type = CatalogRequest.Type.QUERY_SYSTEM,
    description = "Calculate total size of repository"
)
public long calculateRepositorySize(String repositoryId) {
    // Query external system for repository size
    return externalApi.getRepositorySize(repositoryId);
}
```

@ApiRequest

Package: *app.krista.extension.impl.anno.ApiRequest*
Purpose: Defines API request handlers for protocol-specific operations (webhooks, REST endpoints).

Parameters: -
pathElement(): URL path element (required)
name(): Request name (optional)
protocol(): Array of supported protocols (required)

Example:

```
import app.krista.extension.impl.anno.ApiRequest;
import app.krista.extension.request.ProtoRequest;
import app.krista.extension.request.ProtoResponse;

@ApiRequest(
    pathElement = "webhook",
    name = "GitHub Webhook Handler",
    protocol = {"http", "https"}
)
public ProtoResponse handleWebhook(String pathElement, ProtoRequest request)
{
    // Extract webhook payload
```

```
    String payload = request.getBody();
    String eventType = request.getHeader("X-GitHub-Event");

    logger.info("Received GitHub webhook: {}", eventType);

    // Process webhook based on event type
    switch (eventType) {
        case "push" -> processPushEvent(payload);
        case "pull_request" -> processPullRequestEvent(payload);
        case "issues" -> processIssueEvent(payload);
        default -> logger.warn("Unknown event type: {}", eventType);
    }

    return ProtoResponse.success();
}

@ApiRequest(
    pathElement = "status",
    name = "Health Check Endpoint",
    protocol = {"http", "https"}
)
public ProtoResponse healthCheck(String pathElement, ProtoRequest request) {
    Map<String, Object> status = new HashMap<>();
    status.put("status", "healthy");
    status.put("timestamp", System.currentTimeMillis());

    return ProtoResponse.success(status);
}
```

@EntityRequest

Package: *app.krista.extension.impl.anno.EntityRequest*
Purpose: Defines entity CRUD operations for data management.

Parameters: - *type()*: EntityRequest.Type enum value (required)

Entity Request Types:

Type	Purpose	Method Signature	Description
CREATE	Create new entity	*String method(EntityType type, EntityAttributeField attributes)*	Creates entity and returns ID
UPDATE	Update existing entity	*Object method(EntityType type, Object entityId)*	Updates entity by ID
GET	Retrieve entity by ID	*Object method(EntityType type, String entityId)*	Gets single entity
SEARCH	Search entities	*List<Object> method(EntityType type, SearchQuery query, Long*	Searches with pagination

Type	Purpose	Method Signature	Description
		fetchSize, Integer pageNumber)	
DELETE	Delete entity	*void method(EntityType type, String entityId)*	Deletes entity by ID
LOOKUP	Lookup entity by attributes	*Object method(EntityType type, EntityAttributeField attributes, Object context)*	Finds entity by attributes
SUPPORTS	Check operation support	*Boolean method(EntityType type, Object operation, Object context)*	Checks if operation is supported
CONTAINS	Check entity existence	*Boolean method(EntityType type, String entityId)*	Checks if entity exists

Example:

```
import app.krista.extension.impl.anno.EntityRequest;
import app.krista.model.entity.EntityType;
import app.krista.model.entity.EntityAttributeField;
import app.krista.model.entity.SearchQuery;
import java.util.List;

@EntityRequest(type = EntityRequest.Type.CREATE)
public String createRepository(EntityType entityType, EntityAttributeField at
tributes) {
    // Extract attributes
    String repositoryName = attributes.getString("Repository Name");
    boolean isPrivate = attributes.getBoolean("Is Private");

    // Create repository in external system
    String repositoryId = externalApi.createRepository(repositoryName, isPriv
ate);

    // Persist in local storage
    persistRepository(repositoryId, attributes);

    return repositoryId;
}

@EntityRequest(type = EntityRequest.Type.SEARCH)
public List<Object> searchRepositories(EntityType entityType, SearchQuery que
ry,
                                       Long fetchSize, Integer pageNumber) {
    // Build search criteria from query
    String searchTerm = query.getSearchTerm();

    // Execute search with pagination
    return externalApi.searchRepositories(searchTerm, fetchSize, pageNumber);
}
```

```java
@EntityRequest(type = EntityRequest.Type.GET)
public Object getRepository(EntityType entityType, String repositoryId) {
    // Retrieve specific repository
    return externalApi.getRepository(repositoryId);
}

@EntityRequest(type = EntityRequest.Type.UPDATE)
public Object updateRepository(EntityType entityType, Object entityId) {
    String repositoryId = (String) entityId;
    // Update repository in external system
    return externalApi.updateRepository(repositoryId);
}

@EntityRequest(type = EntityRequest.Type.DELETE)
public void deleteRepository(EntityType entityType, String repositoryId) {
    // Delete from external system
    externalApi.deleteRepository(repositoryId);
    // Delete from local storage
    deleteLocalRepository(repositoryId);
}
```

Field Annotations

Field annotations define the configuration attributes for your extension. They appear in the Setup tab and can be applied to extension classes, catalog request methods, and method parameters.

Package: *app.krista.extension.impl.anno.Field*

Available Field Types: -
@Field - Base annotation for custom field types
@Field.Text - Text input field
@Field.Boolean - Checkbox field
@Field.Number - Numeric input field
@Field.PickOne - Dropdown selection field
@Field.Date - Date picker field
@Field.DateRange - Date range picker
@Field.DateTimeRange - Date and time range picker
@Field.Entity - Entity reference field
@Field.File - File upload field
@Field.List - List of values
@Field.MultiField - Complex field with multiple sub-fields

Common Field Parameters: -
name(): Field display name (optional for parameters, required for class-level)
value(): Default value (optional)
order(): Display order (default: -1)

required(): Whether field is required (default: true)
attributes(): Array of @Attribute for additional configuration
options(): Array of @Option for field options

@Field (Base Annotation)

Purpose: Generic field annotation for custom field types and base for specialized field annotations.

Parameters: -
type(): Field type string (required)
name(): Field display name (required)
order(): Display order (default: -1)
required(): Whether field is required (default: true)
attributes(): Array of field attributes (optional)
options(): Array of field options (optional)

Example:

```
@Field(
    type = "Text",
    name = "Custom Field",
    order = 1,
    required = true,
    attributes = {
        @Attribute(name = "visualWidth", value = "M"),
        @Attribute(name = "placeholder", value = "Enter value")
    }
)
```

@Field.Text

Purpose: Text input field with validation and formatting options. Supports password masking with *isSecured* parameter.

Parameters: -
value(): Default value (optional)
name(): Field display name (optional)
order(): Display order (default: -1)
isSecured(): Whether field contains sensitive data - encrypts in database (default: false)
required(): Whether field is required (default: true)
attributes(): Array of field attributes (optional)
options(): Array of field options (optional)

Example:

```
// Regular text field
@Field.Text(
    name = "API URL",
```

```
    required = true,
    attributes = {
        @Attribute(name = "visualWidth", value = "L"),
        @Attribute(name = "placeholder", value = "https://api.example.com")
    }
)

// Secured text field (password, API key, etc.)
@Field.Text(
    name = "API Token",
    required = true,
    isSecured = true,  // Encrypted in database, masked in UI
    attributes = {
        @Attribute(name = "visualWidth", value = "L"),
        @Attribute(name = "inputType", value = "password")
    }
)

// Text field with default value
@Field.Text(
    name = "Host",
    value = "localhost",
    required = false
)
```

@Field.Boolean

Purpose: Boolean checkbox or toggle field.

Parameters: -
value(): Default value (optional)
name(): Field display name (optional)
order(): Display order (default: -1)
required(): Whether field is required (default: true)
attributes(): Array of field attributes (optional)
options(): Array of field options (optional)

Example:

```
@Field.Boolean(
    name = "Enable SSL",
    required = true,
    attributes = {
        @Attribute(name = "defaultValue", value = "true")
    }
)

@Field.Boolean(
    name = "Include Private Repositories",
    required = false,
```

```
    attributes = {
        @Attribute(name = "defaultValue", value = "false")
    }
)
```

@Field.Number

Purpose: Numeric input field for integers and decimals.

Parameters: -
value(): Default value (optional)
name(): Field display name (optional)
order(): Display order (default: -1)
required(): Whether field is required (default: true)
attributes(): Array of field attributes (optional)
options(): Array of field options (optional)

Example:

```
@Field.Number(
    name = "Port",
    required = true,
    attributes = {
        @Attribute(name = "defaultValue", value = "443"),
        @Attribute(name = "min", value = "1"),
        @Attribute(name = "max", value = "65535")
    }
)

@Field.Number(
    name = "Timeout (seconds)",
    required = false,
    attributes = {
        @Attribute(name = "defaultValue", value = "30"),
        @Attribute(name = "min", value = "1"),
        @Attribute(name = "max", value = "300")
    }
)

@Field.Number(
    name = "Max Retries",
    value = "3",
    required = true
)
```

@Field.PickOne

Purpose: Single-selection dropdown or radio button field.

Parameters: -
value(): Default selected value (optional)
name(): Field display name (optional)
values(): Array of available options (optional)
order(): Display order (default: -1)
required(): Whether field is required (default: true)
attributes(): Array of field attributes (optional)
options(): Array of field options (optional)

Example:

```
@Field.PickOne(
    name = "Environment",
    values = {"Production", "Staging", "Development"},
    required = true,
    attributes = {
        @Attribute(name = "defaultValue", value = "Production")
    }
)

@Field.PickOne(
    name = "Log Level",
    values = {"ERROR", "WARN", "INFO", "DEBUG", "TRACE"},
    required = false,
    attributes = {
        @Attribute(name = "defaultValue", value = "INFO")
    }
)

@Field.PickOne(
    name = "Repository Visibility",
    values = {"public", "private", "internal"},
    value = "public",
    required = true
)
```

@Field.Date

Purpose: Date and time picker with constraints.

Parameters: -
value(): Default value (optional)
name(): Field display name (optional)
order(): Display order (default: -1)
required(): Whether field is required (default: false)
includeTimeOfDay(): Include time selection (default: true)
allowPast(): Allow past dates (default: true)
allowToday(): Allow today's date (default: true)

allowFuture(): Allow future dates (default: true)
showHowManyDaysInViewer(): Days to show in viewer (default: 1)
defaultTimeSpan(): Default time span (default: 1)
attributes(): Array of field attributes (optional)
options(): Array of field options (optional)

Example:

```
@Field.Date(
    name = "Start Date",
    required = true,
    includeTimeOfDay = true,
    allowPast = false,
    allowToday = true,
    allowFuture = true,
    attributes = {
        @Attribute(name = "format", value = "yyyy-MM-dd HH:mm:ss")
    }
)

@Field.Date(
    name = "Created After",
    required = false,
    includeTimeOfDay = false,
    allowPast = true,
    allowToday = true,
    allowFuture = false
)
```

@Field.DateRange

Purpose: Date range picker for selecting start and end dates.

Parameters: -
value(): Default value (optional)
name(): Field display name (optional)
order(): Display order (default: -1)
required(): Whether field is required (default: false)
allowPast(): Allow past dates (default: true)
allowToday(): Allow today's date (default: true)
allowFuture(): Allow future dates (default: true)
showHowManyDaysInViewer(): Days to show in viewer (default: 1)
defaultTimeSpan(): Default time span in days (default: 1)
attributes(): Array of field attributes (optional)
options(): Array of field options (optional)

Example:

```
@Field.DateRange(
    name = "Reporting Period",
    required = true,
    allowPast = true,
    allowToday = true,
    allowFuture = false,
    defaultTimeSpan = 7  // Default to 7-day range
)
```

@Field.DateTimeRange

Purpose: Date and time range picker with precise time selection.

Parameters: - Similar to @Field.DateRange with additional: - *`allowNow()`*: Allow current date/time (default: true)

Example:

```
@Field.DateTimeRange(
    name = "Event Time Range",
    required = true,
    allowPast = true,
    allowNow = true,
    allowFuture = true,
    defaultTimeSpan = 1
)
```

@Field.Entity

Purpose: Entity reference field for complex data types.

Parameters: -
`value()`: Default value (optional)
`name()`: Field display name (optional)
`entityName()`: Referenced entity name (optional)
`order()`: Display order (default: -1)
`required()`: Whether field is required (default: true)
`attributes()`: Array of field attributes (optional)
`options()`: Array of field options (optional)

Example:

```
@Field.Entity(
    name = "Default User",
    entityName = "User",
    required = false,
    attributes = {
        @Attribute(name = "selector", value = "entitySelectorForUser")
    }
)
```

```
@Field.Entity(
    name = "Target Repository",
    entityName = "GitHubRepository",
    required = true
)
```

@Field.File

Purpose: File upload field with multiple file support.

Parameters: -
value(): Default value (optional)
name(): Field display name (optional)
order(): Display order (default: -1)
required(): Whether field is required (default: true)
multipleFileUpload(): Allow multiple files (default: false)
attributes(): Array of field attributes (optional)
options(): Array of field options (optional)

Example:

```
@Field.File(
    name = "Certificate File",
    required = true,
    multipleFileUpload = false,
    attributes = {
        @Attribute(name = "acceptedTypes", value = ".pem,.crt,.cer"),
        @Attribute(name = "maxFileSize", value = "10MB")
    }
)

@Field.File(
    name = "Attachments",
    required = false,
    multipleFileUpload = true,
    attributes = {
        @Attribute(name = "acceptedTypes", value = ".pdf,.doc,.docx,.zip"),
        @Attribute(name = "maxFileSize", value = "100MB")
    }
)
```

Dependency Annotation

@Dependency

Package: *app.krista.extension.impl.anno.Dependency*
Purpose: Declares dependencies on other extensions.

Parameters: -
extensionName(): Name of the required extension (required)
domainId(): Domain ID of the required extension (optional)
description(): Description of the dependency (optional)
optional(): Whether dependency is optional (default: false)

Example:

```
@Extension(name = "Advanced GitHub Integration")
@Dependency(
    extensionName = "GitHub Authentication",
    domainId = "catEntryDomain_github-auth-12345",
    description = "Required for user authentication",
    optional = false
)
@Dependency(
    extensionName = "GitHub Webhooks",
    description = "Optional webhook support",
    optional = true
)
public class AdvancedGitHubExtension {
    // Extension implementation
}
```

Entity Annotations

@Entity

Package: *app.krista.extension.impl.anno.Entity*
Purpose: Defines entity metadata for data model classes.

Parameters: -
name(): Entity display name (required)
id(): Unique entity identifier (required)
primaryKey(): Primary key field name (required)
supportStore(): Enable entity storage (required)
options(): Array of entity-level options (optional)

Example:

```
import app.krista.extension.impl.anno.Entity;
import app.krista.extension.impl.anno.Field;
import app.krista.extension.impl.anno.Option;
import app.krista.extension.impl.anno.Attribute;

@Entity(
    name = "GitHub Repository",
    id = "localDomainEntity_github-repo-12345",
    primaryKey = "Repository ID",
    supportStore = true,
```

```
    options = {
        @Option(name = "searchAll", type = "Boolean", value = "true"),
        @Option(name = "searchTimeout", type = "Integer", value = "30000")
    }
)
public class GitHubRepository {
    @Field.Text(name = "Repository ID", required = true,
        attributes = {@Attribute(name = "visualWidth", value = "S")})
    public String repositoryId;

    @Field.Text(name = "Repository Name", required = true,
        attributes = {@Attribute(name = "visualWidth", value = "M")})
    public String name;

    @Field.Boolean(name = "Is Private", required = false)
    public boolean isPrivate;

    @Field.Text(name = "Description", required = false)
    public String description;
}
```

@Classification

Package: *app.krista.extension.impl.anno.Classification*
Purpose: Defines data classification tags for fields or entities for compliance and security.

Parameters: - *value()*: Array of classification strings (required)

Common Classifications: - *"public"*: Publicly accessible data - *"pii"*: Personally Identifiable Information - *"confidential"*: Confidential business data - *"internal"*: Internal use only

Example:

```
import app.krista.extension.impl.anno.Classification;
import app.krista.extension.impl.anno.Field;

@Entity(name = "User Profile", id = "user-profile-entity", primaryKey = "User
ID", supportStore = true)
@Classification({"pii", "confidential"})
public class UserProfile {

    @Field.Text(name = "User ID", required = true)
    @Classification({"public"})
    public String userId;

    @Field.Text(name = "Email Address", required = true)
    @Classification({"pii"})
    public String email;
```

```
    @Field.Text(name = "Social Security Number", required = false, isSecured
= true)
    @Classification({"pii", "confidential"})
    public String ssn;
}
```

@Searchable

Package: *app.krista.extension.impl.anno.Searchable*
Purpose: Enables search functionality for entities or fields with timeout configuration.

Parameters: - *timeout()*: Search timeout in seconds (default: 25)

Example:

```
import app.krista.extension.impl.anno.Searchable;
import app.krista.extension.impl.anno.Entity;
import app.krista.extension.impl.anno.Field;

@Entity(name = "Repository", id = "repo-entity", primaryKey = "ID", supportSt
ore = true)
@Searchable(timeout = 30) // Entity-level searchable with 30-second timeout
public class Repository {

    @Field.Text(name = "Repository ID", required = true)
    public String id;

    @Field.Text(name = "Repository Name", required = true)
    @Searchable(timeout = 15) // Field-level searchable with 15-second timeou
t
    public String name;

    @Field.Text(name = "Description", required = false)
    @Searchable(timeout = 20)
    public String description;
}
```

@ToString

Package: *app.krista.extension.impl.anno.ToString*
Purpose: Marks fields to be included in entity summary/display representations.

Parameters: None

Example:

```
import app.krista.extension.impl.anno.ToString;
import app.krista.extension.impl.anno.Entity;
import app.krista.extension.impl.anno.Field;

@Entity(name = "Repository", id = "repo-entity", primaryKey = "ID", supportSt
```

```
ore = true)
public class Repository {

    @Field.Text(name = "Repository ID", required = true)
    public String id;

    @ToString // Include in summary display
    @Field.Text(name = "Repository Name", required = true)
    public String name;

    @ToString // Include in summary display
    @Field.Text(name = "Owner", required = true)
    public String owner;

    @Field.Text(name = "Description", required = false)
    public String description; // Not included in summary
}
```

Metadata Annotations

@Attribute

Package: *app.krista.extension.impl.anno.Attribute*
Purpose: Defines field or entity attributes for UI rendering and validation.

Parameters:
name(): Attribute name (required)
value(): Attribute value (required)

Common Attributes:
visualWidth: Field width (S, M, L, XL)
placeholder: Placeholder text
defaultValue: Default value
min: Minimum value (for numbers)
max: Maximum value (for numbers)
format: Date/time format
inputType: Input type (text, password, email, url)
acceptedTypes: Accepted file types
maxFileSize: Maximum file size

Example:

```
@Field.Text(
    name = "API URL",
    attributes = {
        @Attribute(name = "visualWidth", value = "L"),
        @Attribute(name = "placeholder", value = "https://api.example.com"),
        @Attribute(name = "inputType", value = "url")
```

```
    }
)
```

@Option

Package: *app.krista.extension.impl.anno.Option*
Purpose: Defines field or entity options for advanced configuration.

Parameters:
name(): Option name (required)
type(): Option type (required)
value(): Option value (required)

Example:

```
@Entity(
    name = "Repository",
    id = "repo-entity",
    primaryKey = "ID",
    supportStore = true,
    options = {
        @Option(name = "searchAll", type = "Boolean", value = "true"),
        @Option(name = "searchTimeout", type = "Integer", value = "30000"),
        @Option(name = "cacheEnabled", type = "Boolean", value = "false")
    }
)
```

Deployment Annotations

@Java

Package: *app.krista.extension.impl.anno.Java*
Purpose: Specifies Java version requirement for the extension.

Parameters: - *version()*: Java.Version enum value (required)

Available Versions: - *Java.Version.JAVA_17*: Java 17 - *Java.Version.JAVA_21*: Java 21 (recommended)

Example:

```
import app.krista.extension.impl.anno.Java;

@Java(version = Java.Version.JAVA_21)
@Extension(
    name = "Modern Extension",
    version = "1.0.0"
)
public class ModernExtension {
    // Can use Java 21 features: pattern matching, records, virtual threads,
```

```
etc.
}
```

@Containerize

Package: *app.krista.extension.impl.anno.Containerize*
Purpose: Configures containerization settings for the extension.

Parameters: -
baseImageVersion(): Base Docker image version (required)
scalable(): Whether extension can scale horizontally (default: true)
deployingInstanceMode(): Deployment mode (default: INVOKER)

Deployment Modes: -
DeployingInstanceMode.INVOKER: One instance per invoker
DeployingInstanceMode.WORKSPACE: One instance per workspace
DeployingInstanceMode.APPLIANCE: One instance per appliance

Example:

```
import app.krista.extension.impl.anno.Containerize;

@Containerize(
    baseImageVersion = "krista-base:21.0",
    scalable = true,
    deployingInstanceMode = Containerize.DeployingInstanceMode.INVOKER
)
@Extension(name = "Containerized Extension")
public class ContainerizedExtension {
    // Extension implementation
}
```

KSDK Classes and Interfaces

The Krista Software Development Kit (KSDK) provides core components and utilities for extension development. All KSDK classes are available in the *krista-service-apis-java/ksdk/* directory.

Core KSDK Components

ExtensionResponse: Standard response object for catalog requests

```
import app.krista.extension.response.ExtensionResponse;

// Success response with data
return ExtensionResponse.success()
    .withData("users", userList)
    .withMetadata("count", userList.size());
```

```
// Error response
return ExtensionResponse.error("Failed to fetch users: " + e.getMessage());

// Response with warnings
return ExtensionResponse.success()
    .withData("result", result)
    .withWarning("Some data may be incomplete");
```

InvokerStore: Interface for saving/validating invoker configuration

```
import app.krista.ksdk.invoker.InvokerStore;

@Inject
private InvokerStore invokerStore;

public void saveConfiguration(Map<String, Object> attributes) {
    invokerStore.updateSetup(attributes);
}
```

AuthorizationContext: Security context for permission checks

```
import app.krista.extension.authorization.AuthorizationContext;

@Inject
private AuthorizationContext authContext;

public boolean hasPermission(String permission) {
    return authContext.hasPermission(permission);
}
```

EventSubscription: Event management for asynchronous processing

```
import app.krista.extension.executor.EventSubscription;

@InvokerRequest(InvokerRequest.Type.EVENT_DELIVERED)
public void handleEvent(EventSubscription subscription, Map<String, Object> e
ventData) {
    String eventType = (String) eventData.get("eventType");
    processEvent(eventType, eventData);
}
```

Dependency Injection with HK2

Extensions use HK2 for dependency injection. Services are automatically injected using *@Inject*:

```
import jakarta.inject.Inject;
import app.krista.ksdk.invoker.InvokerStore;

public class MyExtension {
```

```
    @Inject
    private InvokerStore invokerStore;

    @Inject
    private AuthorizationContext authContext;

    // Services are automatically injected by the platform
}
```

Part III: Core Implementation

Catalog Requests

Catalog requests are the primary way extensions interact with external systems. They represent operations that users can invoke through the Krista platform.

Catalog Request Types

CHANGE_SYSTEM: Modifies external system state - Creating, updating, or deleting resources - Sending notifications or messages - Triggering workflows or processes

QUERY_SYSTEM: Retrieves data without modification - Fetching lists of resources - Searching for specific items - Retrieving configuration or status information

WAIT_FOR_EVENT: Waits for external events - Polling for status changes - Waiting for webhook callbacks - Monitoring for specific conditions

Implementing Catalog Requests

Basic Catalog Request:

```
@CatalogRequest(
    id = "localDomainRequest_get-user",
    name = "Get User",
    description = "Retrieve user information by ID",
    area = "User Management",
    type = CatalogRequest.Type.QUERY_SYSTEM
)
@Field.Text(name = "User ID", required = true)
public ExtensionResponse getUser(
    @Field.Text(name = "User ID") String userId
) {
    try {
        // 1. Call integration layer
        ExternalUser externalUser = apiClient.getUser(userId);
```

```
        // 2. Transform to Krista entity
        KristaUser kristaUser = userTransformer.transform(externalUser);

        // 3. Return success response
        return ExtensionResponse.success()
            .withData("user", kristaUser);

    } catch (ApiException e) {
        return ExtensionResponse.error("Failed to retrieve user: " + e.getMes
sage());
    }
}
```

Catalog Request with Multiple Parameters:

```
@CatalogRequest(
    id = "localDomainRequest_search-users",
    name = "Search Users",
    description = "Search for users matching criteria",
    area = "User Management",
    type = CatalogRequest.Type.QUERY_SYSTEM
)
@Field.Text(name = "Search Query", required = false)
@Field.Number(name = "Max Results", required = false)
@Field.Boolean(name = "Include Inactive", required = false)
public ExtensionResponse searchUsers(
    @Field.Text(name = "Search Query") String query,
    @Field.Number(name = "Max Results") Integer maxResults,
    @Field.Boolean(name = "Include Inactive") Boolean includeInactive
) {
    // Set defaults
    if (maxResults == null) maxResults = 100;
    if (includeInactive == null) includeInactive = false;

    try {
        List<ExternalUser> externalUsers = apiClient.searchUsers(query, maxRe
sults, includeInactive);
        List<KristaUser> kristaUsers = userTransformer.transformList(external
Users);

        return ExtensionResponse.success()
            .withData("users", kristaUsers)
            .withMetadata("count", kristaUsers.size())
            .withMetadata("query", query);

    } catch (ApiException e) {
        return ExtensionResponse.error("Search failed: " + e.getMessage());
    }
}
```

CHANGE_SYSTEM Request:

```
@CatalogRequest(
    id = "localDomainRequest_create-user",
    name = "Create User",
    description = "Create a new user in the external system",
    area = "User Management",
    type = CatalogRequest.Type.CHANGE_SYSTEM
)
@Field.Text(name = "Username", required = true)
@Field.Text(name = "Email", required = true)
@Field.Text(name = "Full Name", required = true)
@Field.Boolean(name = "Send Welcome Email", required = false)
public ExtensionResponse createUser(
    @Field.Text(name = "Username") String username,
    @Field.Text(name = "Email") String email,
    @Field.Text(name = "Full Name") String fullName,
    @Field.Boolean(name = "Send Welcome Email") Boolean sendWelcomeEmail
) {
    try {
        // Validate input
        if (!isValidEmail(email)) {
            return ExtensionResponse.error("Invalid email format");
        }

        // Create user in external system
        ExternalUser newUser = apiClient.createUser(username, email, fullName
);

        // Send welcome email if requested
        if (Boolean.TRUE.equals(sendWelcomeEmail)) {
            emailService.sendWelcomeEmail(email, fullName);
        }

        // Transform and return
        KristaUser kristaUser = userTransformer.transform(newUser);
        return ExtensionResponse.success()
            .withData("user", kristaUser)
            .withMetadata("userId", newUser.getId());

    } catch (ApiException e) {
        return ExtensionResponse.error("Failed to create user: " + e.getMessa
ge());
    }
}
```

Best Practices for Catalog Requests

1. **Always validate input parameters** before calling external systems
2. **Use meaningful error messages** that help users understand what went wrong

3. **Include metadata** in responses for debugging and auditing
4. **Handle null parameters** gracefully with sensible defaults
5. **Log all external system interactions** for troubleshooting
6. **Use appropriate request types** (CHANGE vs QUERY vs WAIT_FOR_EVENT)
7. **Keep request methods focused** - one operation per request
8. **Return structured data** using Krista entities, not raw external data

---

Sub-Catalog Requests

Sub-catalog requests are helper methods that support catalog requests but aren't directly exposed to users. They're useful for breaking down complex operations into smaller, reusable components.

Example:

```
@SubCatalogRequest(
    name = "Validate User Data",
    type = CatalogRequest.Type.QUERY_SYSTEM,
    description = "Validate user data before creation"
)
public boolean validateUserData(String username, String email) {
    if (username == null || username.isBlank()) {
        return false;
    }
    if (!isValidEmail(email)) {
        return false;
    }
    // Check if username already exists
    return !apiClient.userExists(username);
}

@CatalogRequest(
    id = "localDomainRequest_create-user",
    name = "Create User",
    area = "User Management",
    type = CatalogRequest.Type.CHANGE_SYSTEM
)
public ExtensionResponse createUser(String username, String email, String ful
lName) {
    // Use sub-catalog request for validation
    if (!validateUserData(username, email)) {
        return ExtensionResponse.error("Invalid user data");
    }

    // Proceed with user creation
    // ...
}
```

Entity Management

Entities represent external system concepts within the Krista platform. They define the data structures that users interact with and that the platform stores and displays.

Defining Entities with Java Records

Java 21 records are perfect for immutable entity definitions:

```
public record KristaUser(
    String id,
    String username,
    String email,
    String fullName,
    LocalDateTime createdAt,
    boolean isActive
) {
    // Compact constructor for validation
    public KristaUser {
        if (id == null || id.isBlank()) {
            throw new IllegalArgumentException("User ID cannot be blank");
        }
        if (email == null || !isValidEmail(email)) {
            throw new IllegalArgumentException("Invalid email format");
        }
    }

    private static boolean isValidEmail(String email) {
        return email != null && email.matches("^[A-Za-z0-9+_.-]+@(.+)$");
    }
}
```

Entity Best Practices

1. **Use immutable data structures** (Java records or final fields)
2. **Include validation** in constructors or factory methods
3. **Provide meaningful field names** that match business terminology
4. **Document complex fields** with JavaDoc comments
5. **Use appropriate data types** (LocalDateTime for dates, BigDecimal for money, etc.)
6. **Keep entities focused** - one entity per external system concept

Data Transformation Layer

The transformation layer converts between external system data formats and Krista entities. This is a critical component that ensures data consistency and quality.

Transformer Pattern

Basic Transformer:

```
public class UserTransformer {

    public KristaUser transform(ExternalUser externalUser) {
        if (externalUser == null) {
            return null;
        }

        return new KristaUser(
            externalUser.getId(),
            externalUser.getUsername(),
            externalUser.getEmail(),
            externalUser.getFirstName() + " " + externalUser.getLastName(),
            parseDateTime(externalUser.getCreatedAt()),
            "active".equalsIgnoreCase(externalUser.getStatus())
        );
    }

    public List<KristaUser> transformList(List<ExternalUser> externalUsers) {
        if (externalUsers == null) {
            return Collections.emptyList();
        }

        return externalUsers.stream()
            .map(this::transform)
            .filter(Objects::nonNull)
            .toList();
    }

    private LocalDateTime parseDateTime(String dateTimeStr) {
        try {
            return LocalDateTime.parse(dateTimeStr, DateTimeFormatter.ISO_DAT
E_TIME);
        } catch (Exception e) {
            logger.warn("Failed to parse datetime: {}", dateTimeStr);
            return null;
        }
    }
}
```

Bidirectional Transformation

For CHANGE_SYSTEM operations, you need to transform Krista data back to external format:

```
public class UserTransformer {
```

```
    // Krista → External
    public ExternalUser toExternal(KristaUser kristaUser) {
        ExternalUser externalUser = new ExternalUser();
        externalUser.setId(kristaUser.id());
        externalUser.setUsername(kristaUser.username());
        externalUser.setEmail(kristaUser.email());

        // Split full name into first and last
        String[] nameParts = kristaUser.fullName().split(" ", 2);
        externalUser.setFirstName(nameParts[0]);
        externalUser.setLastName(nameParts.length > 1 ? nameParts[1] : "");

        externalUser.setStatus(kristaUser.isActive() ? "active" : "inactive")
;
        return externalUser;
    }

    // External → Krista
    public KristaUser fromExternal(ExternalUser externalUser) {
        return new KristaUser(
            externalUser.getId(),
            externalUser.getUsername(),
            externalUser.getEmail(),
            externalUser.getFirstName() + " " + externalUser.getLastName(),
            parseDateTime(externalUser.getCreatedAt()),
            "active".equalsIgnoreCase(externalUser.getStatus())
        );
    }
}
```

Transformation Best Practices

1. **Handle null values gracefully** - don't let null propagate
2. **Provide default values** for missing data when appropriate
3. **Log transformation failures** with sufficient context
4. **Validate transformed data** before returning
5. **Keep transformers stateless** - no instance variables
6. **Use helper methods** for complex transformations
7. **Test transformations thoroughly** with edge cases

Part IV: Setup Tab Implementation

Setup Tab Overview

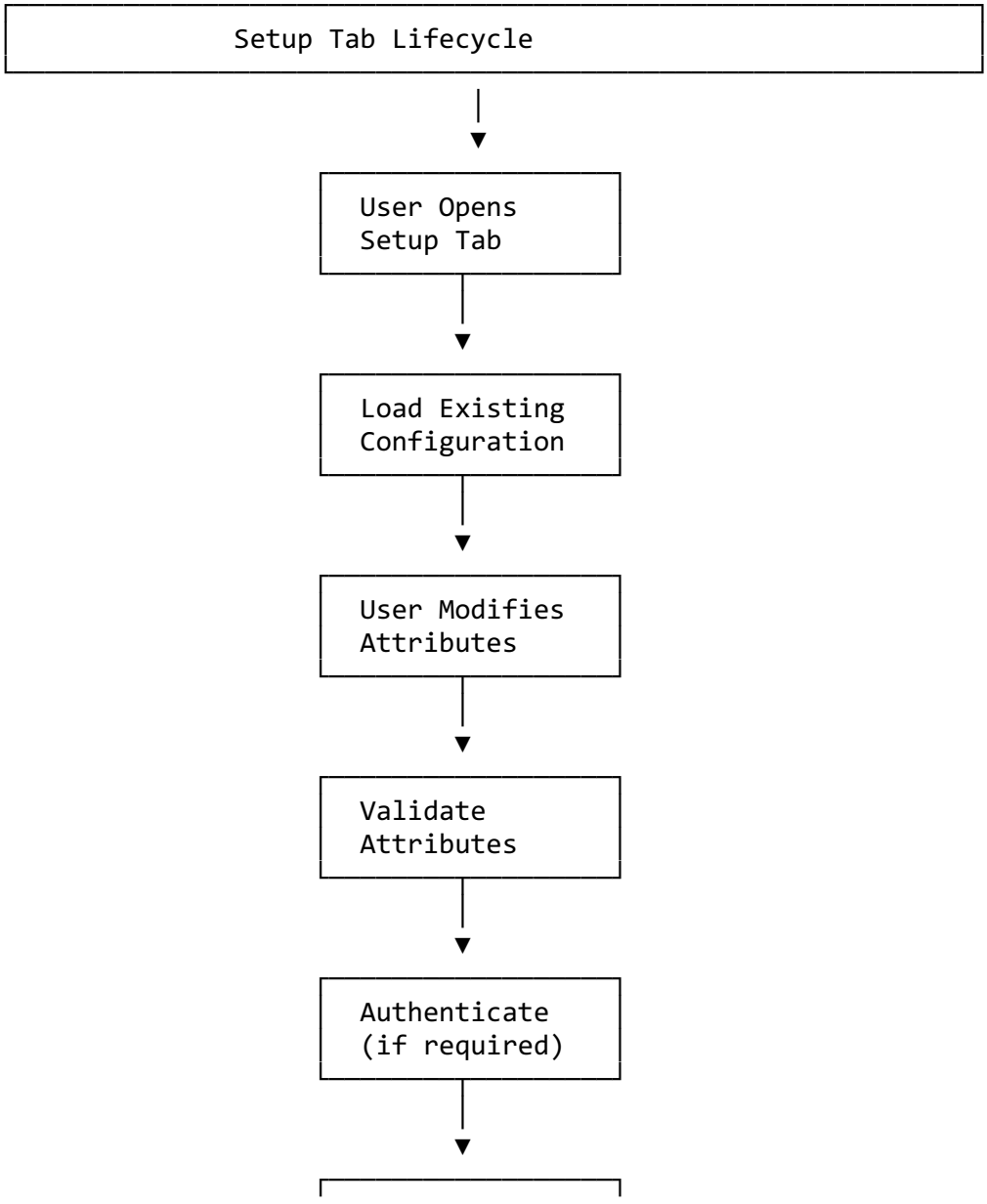
The Setup tab is where users configure your extension before deploying it. This section covers the complete implementation of Setup tab functionality.

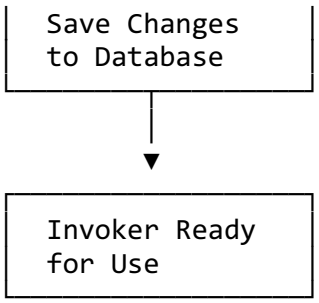
Setup Tab Components

The Setup tab consists of several key components:

1. **Attributes Section**: User-configurable fields (text, numbers, booleans, etc.)
2. **Dependencies Section**: Other extensions this extension requires
3. **Authentication Section**: How users authenticate to use this extension
4. **Validation Logic**: Server-side validation before saving
5. **Trust Configuration**: SSL/TLS certificates for secure connections
6. **Save Mechanism**: Persisting configuration to database

Setup Tab Lifecycle





Technology Stack

- **Frontend**: Angular (TypeScript) - automatically generated from annotations
- **Backend**: Java 11/21 - your extension code
- **Annotations**: Compile-time code generation for UI
- **Dependency Injection**: HK2 framework
- **Database**: PostgreSQL with JSONB for flexible attribute storage
- **Encryption**: AES-256 for secured fields

Attributes Implementation

Overview

Attributes are **configurable parameters** that users set in the Setup tab. They define how your extension behaves at runtime.

Why Attributes Matter

Without attributes: - Extension behavior is hardcoded - Cannot adapt to different environments - Requires code changes for configuration updates

With attributes: - Users configure extension behavior through UI - Same extension works in different environments - No code changes needed for configuration updates

Attribute Types

Type	Description	Example Use Case
Text	String values	API keys, URLs, email addresses
Text (Secured)	Encrypted strings	Passwords, secret tokens
Number	Numeric values	Port numbers, timeouts, limits
Boolean	True/false flags	Enable/disable features
PickOne	Dropdown selection	Environment selection, log levels
Date	Date/time values	Expiration dates, scheduled times
Entity	Reference to entity	User selection, product reference

Basic Attribute Definition

Example from kristagrand:

```
@Field.Text(name = "Account Email")
public final class WsCertificationExtension {
```

Why this works: - *@Field.Text(name = "Account Email")*: Defines a text field in the Setup tab - The field appears as a text input box labeled “Account Email” - Users can enter an email address which is stored in the database

Text Attributes

Text attributes are the most common type. They store string values.

Modern syntax:

```
@Field.Text(name = "API Key", required = true)
@Field.Text(name = "Host URL", required = true)
@Field.Text(name = "Description", required = false)
```

Secured Text Attributes (Passwords)

For sensitive data like passwords and API tokens, use the *isSecured* parameter to encrypt the value.

⚠ **SECURITY WARNING**: Always use *isSecured = true* for passwords, API keys, and other sensitive data!

```
@Field.Text(name = "API Secret", required = true, isSecured = true)
@Field.Text(name = "Database Password", required = true, isSecured = true)
```

What happens: - Value is encrypted before storing in database - Decrypted only when extension needs it at runtime - Never exposed in logs or API responses - UI shows password input field (masked characters)

Number Attributes

Number attributes store numeric values (integers or decimals).

```
@Field.Number(name = "Port", required = true)
@Field.Number(name = "Timeout Seconds", required = false)
@Field.Number(name = "Max Retries", required = true)
```

Use cases: - Port numbers (e.g., 8080, 443) - Timeout values in seconds - Retry counts - Limits and thresholds

Note: Number fields in Krista use *Double* type to support both integers and decimals.

Boolean Attributes

Boolean attributes are true/false flags for enabling/disabling features.

```
@Field.Boolean(name = "Enable Logging", required = false)
@Field.Boolean(name = "Use SSL", required = true)
@Field.Boolean(name = "Auto Retry", required = false)
```

Default behavior: - If *`required = false`* and user doesn't set value, defaults to *`false`* - Renders as checkbox in UI - Stored as boolean in database

PickOne Attributes (Dropdown)

PickOne attributes provide a dropdown list of options.

```
@Field.PickOne(
    name = "Environment",
    required = true,
    values = {"Development", "Staging", "Production"}
)

@Field.PickOne(
    name = "Log Level",
    required = false,
    values = {"DEBUG", "INFO", "WARN", "ERROR"}
)
```

Use cases: - Environment selection (dev, staging, prod) - Log level selection (DEBUG, INFO, WARN, ERROR) - Protocol selection (HTTP, HTTPS) - Region selection

Date Attributes

Date attributes store timestamps.

```
@Field.Date(name = "Expiration Date", required = false)
@Field.Date(name = "Start Time", required = true)
```

Storage: - Stored as *Long* (Unix timestamp in milliseconds) - UI provides date/time picker - Can include time-of-day or just date

Entity Attributes

Entity attributes reference other entities in the system.

```
@Field.Entity(name = "Default User", required = false, entityName = "User")
@Field.Entity(name = "Target Account", required = true, entityName = "Account
")
```

Use cases: - Selecting a default user - Referencing a product or category - Linking to another entity

Multiple Attributes Example

Real-world extensions typically have multiple attributes:

```
@Field(name = GuestAuthenticationExtension.DEFAULT_ROLE_KEY, type = "Text", r
equired = false)
@Field(name = CommonInvokerParameters.SESSION_TIMEOUT, type = "Number", requi
red = false)
```

Modern equivalent:

```
@Field.Text(name = "Default Role", required = false)
@Field.Number(name = "Session Timeout", required = false)
@Field.Text(name = "Attribute Parameters", required = false)
```

Accessing Attributes at Runtime

Once attributes are configured, your extension needs to access them at runtime.

Pattern: Via InvokerRequest Methods

Attributes are passed as a *Map<String, Object>* parameter:

```
@InvokerRequest(InvokerRequest.Type.VALIDATE_ATTRIBUTES)
public List<String> validateAttributes(Map<String, Object> attributes) {
    List<String> errors = new ArrayList<>();

    // Access text attribute
    String apiKey = (String) attributes.get("API Key");
    if (apiKey == null || apiKey.isBlank()) {
        errors.add("API Key is required");
    }

    // Access number attribute
    Double timeout = (Double) attributes.get("Timeout Seconds");
    if (timeout != null && timeout < 1) {
        errors.add("Timeout must be at least 1 second");
    }

    // Access boolean attribute
    Boolean useSSL = (Boolean) attributes.get("Use SSL");
    if (Boolean.TRUE.equals(useSSL)) {
        // SSL is enabled
    }

    return errors;
}
```

Attribute Best Practices

1. **Use descriptive names**: “API Key” is better than “key”
2. **Mark sensitive fields as secured**: Always use *`isSecured = true`* for passwords
3. **Provide sensible defaults**: Use *`required = false`* with default values when appropriate
4. **Validate thoroughly**: Check for null, empty, and invalid values
5. **Document expected formats**: Use placeholders or descriptions
6. **Group related attributes**: Use consistent naming (e.g., “Database Host”, “Database Port”, “Database Name”)

Dependencies Implementation

Overview

Dependencies allow your extension to use functionality from other extensions. This enables modular architecture where extensions build upon each other.

Declaring Dependencies

Annotation: *`@Dependency`* from *`krista-service-apis-java/extension/impl/anno/src/main/java/app/krista/extension/impl/anno/Dependency.java`*

Parameters: - *`name`*: Logical name for the dependency (e.g., “Authentication”) - *`domainId`*: The domain ID that the dependency must implement - *`description`*: Human-readable description of why this dependency is needed

Real-World Example

```
@Dependency(name = "Authentication", domainId = "catEntryDomain_db053e8f-a194-4dde-aa6f-701ef7a6b3a7",
        description = "Extension dependency for authentication.")
```

What this means: - The Client API extension requires an Authentication extension - The dependency is named “Authentication” (used for injection) - The dependency must implement domain *`catEntryDomain_db053e8f-a194-4dde-aa6f-701ef7a6b3a7`* - The description explains why this dependency is needed

Accessing Dependencies

Once declared, dependencies are injected via constructor using HK2.

```
@Inject
public ClientApiExtension(@Named("self") Invoker invoker,
        @Named("Authentication") Invoker authenticatorInvoker,
        ClientApiRequestAuthenticator requestAuthenticator) {
    this.invoker = invoker;
    this.authenticatorInvoker = authenticatorInvoker;
    this.requestAuthenticator = requestAuthenticator;
}
```

Key points: - *`@Named("self")`*: Injects the current extension's Invoker - *`@Named("Authentication")`*: Injects the dependency by name (matches *`@Dependency(name = "Authentication")`*) - Dependencies are automatically injected by the HK2 framework

Multiple Dependencies

Extensions can have multiple dependencies:

```
@Dependency(name = "Authentication",
            domainId = "catEntryDomain_db053e8f-a194-4dde-aa6f-701ef7a6b3a7",
            description = "User authentication")
@Dependency(name = "Logging",
            domainId = "catEntryDomain_12345678-1234-1234-1234-123456789012",
            description = "Centralized logging")
@Extension(...)
public class MyExtension {
    @Inject
    public MyExtension(
            @Named("Authentication") Invoker authInvoker,
            @Named("Logging") Invoker loggingInvoker) {
        // Use dependencies
    }
}
```

Dependency Best Practices

1. **Minimize dependencies**: Only depend on what you truly need
2. **Use standard domains**: Prefer well-known domains (Authentication, Storage)
3. **Document dependencies**: Explain why each dependency is needed
4. **Avoid circular dependencies**: Design clear hierarchy
5. **Version compatibility**: Ensure dependency versions are compatible

Common Pitfalls

❌ Pitfall: Name mismatch

```
@Dependency(name = "Authentication", domainId = "...")
```

```
@Inject
public MyExtension(@Named("Auth") Invoker authInvoker) {  // Wrong name!
    // authInvoker will be null
}
```

✅ **Correct:**

```
@Inject
public MyExtension(@Named("Authentication") Invoker authInvoker) {  // Matche
s dependency name
}
```

Authentication Implementation

Overview

Authentication determines **who can access your extension** and **how they prove their identity**. Krista supports custom authentication mechanisms through the *`RequestAuthenticator`* interface.

Why Authentication Matters

Without authentication: - Anyone can access your extension - No way to track who performed actions - Security vulnerabilities - Cannot enforce permissions

With authentication: - Verify user identity before granting access - Track actions to specific users - Enforce role-based permissions - Secure sensitive operations

RequestAuthenticator Interface

File: *`krista-service-apis-java/extension/authorization/api/src/main/java/app/krista/extension/authorization/RequestAuthenticator.java`*

The *`RequestAuthenticator`* interface defines how your extension authenticates users.

Implementing Custom Authentication

Example from kristagrand:

```
public class ExtensionRequestAuthenticator implements RequestAuthenticator {
    @Override
    public AuthenticationResult authenticate(AuthenticationRequest request) {
        // Custom authentication logic
    }
}
```

Registering the Authenticator

Use the *@InvokerRequest* annotation with *AUTHENTICATOR* type:

```
@InvokerRequest(InvokerRequest.Type.AUTHENTICATOR)
public RequestAuthenticator getAuthenticator() {
    return new ExtensionRequestAuthenticator();
}
```

Authentication Patterns

Pattern 1: API Key Authentication

```
public class ApiKeyAuthenticator implements RequestAuthenticator {
    @Override
    public AuthenticationResult authenticate(AuthenticationRequest request) {
        String apiKey = request.getCredential("apiKey");

        if (apiKey == null || !isValidApiKey(apiKey)) {
            return AuthenticationResult.failure("Invalid API key");
        }

        return AuthenticationResult.success()
            .withUserId(getUserIdFromApiKey(apiKey))
            .withRoles(getRolesForApiKey(apiKey));
    }
}
```

Pattern 2: Username/Password Authentication

```
public class PasswordAuthenticator implements RequestAuthenticator {
    @Override
    public AuthenticationResult authenticate(AuthenticationRequest request) {
        String username = request.getCredential("username");
        String password = request.getCredential("password");

        if (!validateCredentials(username, password)) {
            return AuthenticationResult.failure("Invalid credentials");
        }

        return AuthenticationResult.success()
            .withUserId(username)
            .withRoles(getRolesForUser(username));
    }
}
```

Pattern 3: OAuth2 Token Authentication

```
public class OAuth2Authenticator implements RequestAuthenticator {
    @Override
    public AuthenticationResult authenticate(AuthenticationRequest request) {
        String accessToken = request.getCredential("accessToken");

        try {
            TokenInfo tokenInfo = validateToken(accessToken);

            return AuthenticationResult.success()
                .withUserId(tokenInfo.getUserId())
                .withRoles(tokenInfo.getRoles())
                .withMetadata("tokenExpiry", tokenInfo.getExpiry());

        } catch (TokenValidationException e) {
            return AuthenticationResult.failure("Invalid or expired token");
        }
    }
}
```

Authentication Best Practices

1. **Never log credentials**: Don't log passwords or tokens
2. **Use secure storage**: Store credentials encrypted
3. **Implement rate limiting**: Prevent brute force attacks
4. **Validate thoroughly**: Check all credential components
5. **Return meaningful errors**: Help users understand authentication failures
6. **Support session management**: Cache authentication results when appropriate

---

Validate Attributes Implementation

Overview

Attribute validation ensures that user-provided configuration is correct before saving. This prevents runtime errors and provides immediate feedback to users.

Why Validation Matters

Without validation: - Invalid configuration causes runtime errors - Users don't know what went wrong - Debugging is difficult - Extension may fail silently

With validation: - Catch errors before saving - Provide clear error messages - Guide users to correct configuration - Prevent invalid states

Implementing Validation

Use the *`@InvokerRequest`* annotation with *`VALIDATE_ATTRIBUTES`* type:

```
@InvokerRequest(InvokerRequest.Type.VALIDATE_ATTRIBUTES)
public List<String> validateAttributes(Map<String, Object> attributes) {
    List<String> errors = new ArrayList<>();

    // Validate API Key
    String apiKey = (String) attributes.get("API Key");
    if (apiKey == null || apiKey.isBlank()) {
        errors.add("API Key is required");
    } else if (apiKey.length() < 32) {
        errors.add("API Key must be at least 32 characters");
    }

    // Validate Host URL
    String hostUrl = (String) attributes.get("Host URL");
    if (hostUrl == null || hostUrl.isBlank()) {
        errors.add("Host URL is required");
    } else if (!isValidUrl(hostUrl)) {
        errors.add("Host URL must be a valid URL (e.g., https://api.example.c
om)");
    }

    // Validate Port
    Double port = (Double) attributes.get("Port");
    if (port != null && (port < 1 || port > 65535)) {
        errors.add("Port must be between 1 and 65535");
    }

    // Validate Boolean combinations
    Boolean useSSL = (Boolean) attributes.get("Use SSL");
    if (Boolean.TRUE.equals(useSSL) && hostUrl != null && !hostUrl.startsWith
("https://")) {
        errors.add("When SSL is enabled, Host URL must use HTTPS");
    }

    return errors;
}

private boolean isValidUrl(String url) {
    try {
        new URL(url);
        return true;
    } catch (MalformedURLException e) {
        return false;
    }
}
```

Validation Patterns

Pattern 1: Required Field Validation

```
String value = (String) attributes.get("Field Name");
if (value == null || value.isBlank()) {
    errors.add("Field Name is required");
}
```

Pattern 2: Format Validation

```
String email = (String) attributes.get("Email");
if (email != null && !email.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
    errors.add("Email must be a valid email address");
}
```

Pattern 3: Range Validation

```
Double timeout = (Double) attributes.get("Timeout");
if (timeout != null && (timeout < 1 || timeout > 300)) {
    errors.add("Timeout must be between 1 and 300 seconds");
}
```

Pattern 4: Conditional Validation

```
Boolean enableFeature = (Boolean) attributes.get("Enable Feature");
String featureConfig = (String) attributes.get("Feature Config");
if (Boolean.TRUE.equals(enableFeature) && (featureConfig == null || featureCo
nfig.isBlank())) {
    errors.add("Feature Config is required when Enable Feature is checked");
}
```

Pattern 5: External System Validation

```
String apiKey = (String) attributes.get("API Key");
if (apiKey != null && !apiKey.isBlank()) {
    try {
        // Test API key by making a test call
        apiClient.testConnection(apiKey);
    } catch (ApiException e) {
        errors.add("API Key is invalid: " + e.getMessage());
    }
}
```

Validation Best Practices

1. **Validate early**: Check all required fields first
2. **Provide specific errors**: “API Key must be at least 32 characters” not “Invalid API Key”
3. **Test external connections**: Validate API keys by testing them
4. **Check combinations**: Validate field interdependencies
5. **Return all errors**: Don’t stop at first error - return complete list
6. **Use consistent error format**: Make errors easy to parse and display

---

Trust Extension Implementation

Overview

Trust configuration handles **SSL/TLS certificates and secure connections** for extensions that communicate with external services. This ensures encrypted, authenticated communication.

Why Trust Configuration Matters

Without trust configuration: - Cannot verify server identity - Vulnerable to man-in-the-middle attacks - Cannot connect to servers with self-signed certificates - Security warnings and connection failures

With trust configuration: - Verify server identity via certificates - Encrypted communication (SSL/TLS) - Support for custom/self-signed certificates - Secure connections to external services

SSL Context Initialization

Example from kristagrand:

```
public static SSLContext createSSLContext(byte[] certBytes, char[] password)
        throws Exception {
    // Load certificate into KeyStore
    final KeyStore ks = KeyStore.getInstance("PKCS12");
    ks.load(new ByteArrayInputStream(certBytes), password);

    // Initialize KeyManagerFactory
    final KeyManagerFactory kmf = KeyManagerFactory.getInstance(
        KeyManagerFactory.getDefaultAlgorithm());
    kmf.init(ks, password);

    // Initialize TrustManagerFactory
    final TrustManagerFactory tmf = TrustManagerFactory.getInstance(
        TrustManagerFactory.getDefaultAlgorithm());
    tmf.init(ks);

    // Create SSL context
    final SSLContext sslContext = SSLContext.getInstance("TLS");
    sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

    return sslContext;
}
```

What this does:
1. **Load certificate**: Reads PKCS12 certificate file
2. **Initialize KeyManager**: Manages private keys for authentication
3. **Initialize TrustManager**: Validates server certificates
4. **Create SSL context**: Combines key and trust managers
5. **Return context**: Used for all SSL/TLS connections

Why PKCS12? - Industry standard format - Supports both certificates and private keys - Password-protected - Cross-platform compatible

Trust Store Configuration

Example from kristagrand:

```
{
  "ssl": {
    "enabled": true,
    "port": 8443,
    "keyStore": {
      "type": "PKCS12",
      "path": "/opt/krista/certs/server.p12",
      "password": "${SSL_KEYSTORE_PASSWORD}"
    },
    "trustStore": {
      "type": "PKCS12",
      "path": "/opt/krista/certs/truststore.p12",
      "password": "${SSL_TRUSTSTORE_PASSWORD}"
    }
  }
}
```

Configuration elements: - *enabled*: Enable/disable SSL - *port*: SSL port (typically 8443 or 443) - *keyStore*: Server's certificate and private key - *trustStore*: Trusted CA certificates - *password*: Encrypted password (from environment variable)

Certificate Loading Pattern

```
public void loadCertificate(String certPath, String password) throws Exceptio
n {
    // Read certificate file
    byte[] certBytes = Files.readAllBytes(Paths.get(certPath));

    // Create SSL context
    SSLContext sslContext = MixSSL.createSSLContext(
        certBytes,
        password.toCharArray()
```

```
    );

    // Use SSL context for connections
    SSLSocketFactory socketFactory = sslContext.getSocketFactory();

    // Create secure connection
    SSLSocket socket = (SSLSocket) socketFactory.createSocket(host, port);
}
```

Custom Trust Manager

For custom certificate validation:

```
public class CustomTrustManager implements X509TrustManager {

    private final X509TrustManager defaultTrustManager;
    private final KeyStore customTrustStore;

    public CustomTrustManager(KeyStore customTrustStore) throws Exception {
        this.customTrustStore = customTrustStore;

        // Initialize default trust manager
        TrustManagerFactory tmf = TrustManagerFactory.getInstance(
            TrustManagerFactory.getDefaultAlgorithm());
        tmf.init((KeyStore) null);
        this.defaultTrustManager = (X509TrustManager) tmf.getTrustManagers()[
0];
    }

    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        try {
            defaultTrustManager.checkClientTrusted(chain, authType);
        } catch (CertificateException e) {
            // Check against custom trust store
            checkCustomTrust(chain, authType);
        }
    }

    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        try {
            defaultTrustManager.checkServerTrusted(chain, authType);
        } catch (CertificateException e) {
            // Check against custom trust store
            checkCustomTrust(chain, authType);
        }
    }
```

```
    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return defaultTrustManager.getAcceptedIssuers();
    }

    private void checkCustomTrust(X509Certificate[] chain, String authType)
            throws CertificateException {
        // Validate against custom trust store
        // Throw CertificateException if validation fails
    }
}
```

SSL/TLS Best Practices

1. **Use strong protocols**: TLS 1.2 or higher
2. **Validate certificates**: Always verify server identity
3. **Secure password storage**: Never hardcode passwords
4. **Regular certificate rotation**: Update certificates before expiration
5. **Monitor certificate expiry**: Alert before certificates expire
6. **Use hostname verification**: Prevent man-in-the-middle attacks

---

Save Changes Implementation

Overview

The Save Changes mechanism persists extension configuration to the database. This includes encrypting secured fields and validating data before storage.

Why Save Changes Matters

Without proper save implementation: - Configuration changes are lost - Secured fields stored in plaintext - No validation before saving - Inconsistent state

With proper save implementation: - Configuration persisted reliably - Secured fields encrypted - Validation before saving - Consistent, recoverable state

InvokerStore Interface

The *InvokerStore* interface provides methods for saving and validating invoker configuration.

File:
kristagrand/servers/src/main/java/com/krista/services/ksdk/invokers/InvokerStoreImpl.java

Save Implementation Pattern

Using FnInvokerUpdateSetup:

```
public class FnInvokerUpdateSetup {
    // Updates invoker setup with encryption for secured fields
}
```

Implementation pattern:

```
@Inject
private InvokerStore invokerStore;

public void saveConfiguration(Map<String, Object> attributes) {
    try {
        // 1. Validate attributes
        List<String> errors = validateAttributes(attributes);
        if (!errors.isEmpty()) {
            throw new ValidationException("Invalid configuration: " + String.
join(", ", errors));
        }

        // 2. Encrypt secured fields
        Map<String, Object> encryptedAttributes = encryptSecuredFields(attrib
utes);

        // 3. Save to database
        invokerStore.updateSetup(encryptedAttributes);

        logger.info("Configuration saved successfully");

    } catch (Exception e) {
        logger.error("Failed to save configuration", e);
        throw new RuntimeException("Failed to save configuration: " + e.getMe
ssage(), e);
    }
}

private Map<String, Object> encryptSecuredFields(Map<String, Object> attribut
es) {
    Map<String, Object> encrypted = new HashMap<>(attributes);

    // Encrypt secured fields (API keys, passwords, etc.)
    if (encrypted.containsKey("API Secret")) {
        String secret = (String) encrypted.get("API Secret");
        encrypted.put("API Secret", encryptValue(secret));
    }
```

```
    if (encrypted.containsKey("Database Password")) {
        String password = (String) encrypted.get("Database Password");
        encrypted.put("Database Password", encryptValue(password));
    }

    return encrypted;
}

private String encryptValue(String value) {
    // Use platform encryption service
    // AES-256 encryption with platform-managed keys
    return encryptionService.encrypt(value);
}
```

Loading Configuration

When loading configuration, decrypt secured fields:

```
public Map<String, Object> loadConfiguration() {
    try {
        // 1. Load from database
        Map<String, Object> attributes = invokerStore.getSetup();

        // 2. Decrypt secured fields
        Map<String, Object> decrypted = decryptSecuredFields(attributes);

        return decrypted;

    } catch (Exception e) {
        logger.error("Failed to load configuration", e);
        throw new RuntimeException("Failed to load configuration: " + e.getMe
ssage(), e);
    }
}

private Map<String, Object> decryptSecuredFields(Map<String, Object> attribut
es) {
    Map<String, Object> decrypted = new HashMap<>(attributes);

    // Decrypt secured fields
    if (decrypted.containsKey("API Secret")) {
        String encrypted = (String) decrypted.get("API Secret");
        decrypted.put("API Secret", decryptValue(encrypted));
    }

    if (decrypted.containsKey("Database Password")) {
        String encrypted = (String) decrypted.get("Database Password");
        decrypted.put("Database Password", decryptValue(encrypted));
    }
```

```
    return decrypted;
}

private String decryptValue(String encryptedValue) {
    // Use platform decryption service
    return encryptionService.decrypt(encryptedValue);
}
```

Database Storage

Configuration is stored in PostgreSQL using JSONB format:

```
CREATE TABLE invoker_config (
    invoker_id UUID PRIMARY KEY,
    setup_attributes JSONB NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Example stored data
{
    "API Key": "encrypted_value_here",
    "Host URL": "https://api.example.com",
    "Port": 443,
    "Use SSL": true,
    "Environment": "Production"
}
```

Save Changes Best Practices

1. **Always validate before saving**: Use *VALIDATE_ATTRIBUTES* handler
2. **Encrypt secured fields**: Never store passwords in plaintext
3. **Use transactions**: Ensure atomic updates
4. **Log changes**: Audit trail for configuration changes
5. **Handle errors gracefully**: Provide clear error messages
6. **Test encryption/decryption**: Verify secured fields work correctly

---

Part V: Advanced Features

Invoker Management and Lifecycle

Overview

An **Invoker** is a runtime instance of an extension with specific configuration. Understanding the invoker lifecycle is critical for proper resource management and event handling.

Invoker Lifecycle Events

The platform provides lifecycle events that your extension can handle:

1. **INVOKER_LOADED**: Called when invoker is first created
2. **INVOKER_UPDATED**: Called when configuration changes
3. **INVOKER_UNLOADED**: Called when invoker is destroyed (implicit)

Implementing Lifecycle Handlers

```
@Extension(...)
public class MyExtension {

    private volatile boolean initialized = false;
    private HttpClient httpClient;

    @InvokerRequest(InvokerRequest.Type.INVOKER_LOADED)
    public void onInvokerLoaded(Map<String, Object> attributes) {
        logger.info("Invoker loaded with attributes: {}", attributes);

        // Initialize resources
        String apiUrl = (String) attributes.get("API URL");
        String apiKey = (String) attributes.get("API Key");

        this.httpClient = createHttpClient(apiUrl, apiKey);
        this.initialized = true;

        logger.info("Extension initialized successfully");
    }

    @InvokerRequest(InvokerRequest.Type.INVOKER_UPDATED)
    public void onInvokerUpdated(Map<String, Object> attributes) {
        logger.info("Invoker configuration updated");

        // Clean up old resources
        if (httpClient != null) {
            httpClient.close();
        }
```

```
        // Reinitialize with new configuration
        String apiUrl = (String) attributes.get("API URL");
        String apiKey = (String) attributes.get("API Key");

        this.httpClient = createHttpClient(apiUrl, apiKey);

        logger.info("Extension reinitialized with new configuration");
    }

    private HttpClient createHttpClient(String apiUrl, String apiKey) {
        return HttpClient.newBuilder()
            .baseUrl(apiUrl)
            .header("Authorization", "Bearer " + apiKey)
            .connectTimeout(Duration.ofSeconds(30))
            .build();
    }
}
```

Resource Management Best Practices

1. **Initialize in INVOKER_LOADED**: Set up connections, caches, thread pools
2. **Clean up in INVOKER_UPDATED**: Close old connections before creating new ones
3. **Use try-with-resources**: For automatic resource cleanup
4. **Handle initialization failures**: Log errors and provide meaningful messages
5. **Thread safety**: Use volatile or synchronized for shared state

Invoker State Management

```
public class StatefulExtension {

    private final AtomicReference<ExtensionState> state = new AtomicReference
<>();

    private record ExtensionState(
        String apiUrl,
        String apiKey,
        HttpClient client,
        LocalDateTime initializedAt
    ) {}

    @InvokerRequest(InvokerRequest.Type.INVOKER_LOADED)
    public void onInvokerLoaded(Map<String, Object> attributes) {
        ExtensionState newState = new ExtensionState(
            (String) attributes.get("API URL"),
            (String) attributes.get("API Key"),
            createHttpClient(attributes),
```

```
            LocalDateTime.now()
        );

        state.set(newState);
    }

    @InvokerRequest(InvokerRequest.Type.INVOKER_UPDATED)
    public void onInvokerUpdated(Map<String, Object> attributes) {
        ExtensionState oldState = state.get();

        // Close old client
        if (oldState != null && oldState.client() != null) {
            oldState.client().close();
        }

        // Create new state
        ExtensionState newState = new ExtensionState(
            (String) attributes.get("API URL"),
            (String) attributes.get("API Key"),
            createHttpClient(attributes),
            LocalDateTime.now()
        );

        state.set(newState);
    }

    @CatalogRequest(CatalogRequest.Type.QUERY_SYSTEM)
    public List<Entity> queryData() {
        ExtensionState currentState = state.get();
        if (currentState == null) {
            throw new IllegalStateException("Extension not initialized");
        }

        return currentState.client().query("/api/data");
    }
}
```

Adding Custom Tabs to Extensions

Overview

Custom tabs allow you to add additional UI pages to your extension beyond the standard Setup tab. This is useful for dashboards, reports, or specialized configuration interfaces.

Implementing Custom Tabs

```
@InvokerRequest(InvokerRequest.Type.CUSTOM_TABS)
public List<CustomTab> getCustomTabs() {
```

```
    return List.of(
        new CustomTab("Dashboard", "/dashboard"),
        new CustomTab("Reports", "/reports"),
        new CustomTab("Advanced Settings", "/advanced")
    );
}
```

Custom Tab Best Practices

1. **Keep tabs focused**: Each tab should have a clear purpose
2. **Use consistent naming**: Match platform conventions
3. **Provide navigation**: Help users understand tab relationships
4. **Handle permissions**: Check user roles before displaying sensitive tabs

---

Event-Based Programming

Overview

Extensions can respond to platform events and external system events. This enables reactive, event-driven architectures.

Wait for Event Pattern

```
@CatalogRequest(CatalogRequest.Type.WAIT_FOR_EVENT)
public Event waitForEvent(Map<String, Object> parameters) {
    String eventType = (String) parameters.get("eventType");
    Long timeout = (Long) parameters.get("timeout");

    // Poll external system for events
    Event event = pollForEvent(eventType, timeout);

    return event;
}

private Event pollForEvent(String eventType, Long timeout) {
    long startTime = System.currentTimeMillis();
    long endTime = startTime + timeout;

    while (System.currentTimeMillis() < endTime) {
        // Check for event
        Event event = checkExternalSystem(eventType);
        if (event != null) {
            return event;
        }

        // Wait before next poll
        try {
```

```
            Thread.sleep(1000); // Poll every second
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Event polling interrupted", e);
        }
    }

    throw new TimeoutException("No event received within timeout period");
}
```

Event Best Practices

1. **Respect timeouts**: Don't block indefinitely
2. **Use efficient polling**: Balance responsiveness with resource usage
3. **Handle interruptions**: Support graceful cancellation
4. **Log event processing**: Track event flow for debugging

Service Integration Patterns

Overview

Extensions integrate with external services using various patterns. Choose the right pattern based on your requirements.

Pattern 1: REST API Integration

```
public class RestApiIntegration {

    private final HttpClient httpClient;
    private final String baseUrl;
    private final String apiKey;

    public RestApiIntegration(String baseUrl, String apiKey) {
        this.baseUrl = baseUrl;
        this.apiKey = apiKey;
        this.httpClient = HttpClient.newBuilder()
            .connectTimeout(Duration.ofSeconds(30))
            .build();
    }

    public <T> T get(String path, Class<T> responseType) {
        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(baseUrl + path))
            .header("Authorization", "Bearer " + apiKey)
            .header("Accept", "application/json")
            .GET()
            .build();
```

```
        try {
            HttpResponse<String> response = httpClient.send(
                request,
                HttpResponse.BodyHandlers.ofString()
            );

            if (response.statusCode() != 200) {
                throw new ApiException("API returned status " + response.stat
usCode());
            }

            return objectMapper.readValue(response.body(), responseType);

        } catch (Exception e) {
            throw new RuntimeException("API call failed", e);
        }
    }

    public <T> T post(String path, Object body, Class<T> responseType) {
        String jsonBody = objectMapper.writeValueAsString(body);

        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(baseUrl + path))
            .header("Authorization", "Bearer " + apiKey)
            .header("Content-Type", "application/json")
            .POST(HttpRequest.BodyPublishers.ofString(jsonBody))
            .build();

        try {
            HttpResponse<String> response = httpClient.send(
                request,
                HttpResponse.BodyHandlers.ofString()
            );

            if (response.statusCode() != 200 && response.statusCode() != 201)
{
                throw new ApiException("API returned status " + response.stat
usCode());
            }

            return objectMapper.readValue(response.body(), responseType);

        } catch (Exception e) {
            throw new RuntimeException("API call failed", e);
        }
    }
}
```

Pattern 2: Retry with Exponential Backoff

```
public class RetryableApiClient {

    private static final int MAX_RETRIES = 3;
    private static final long INITIAL_BACKOFF_MS = 1000;

    public <T> T executeWithRetry(Supplier<T> operation) {
        int attempt = 0;
        long backoff = INITIAL_BACKOFF_MS;

        while (true) {
            try {
                return operation.get();

            } catch (Exception e) {
                attempt++;

                if (attempt >= MAX_RETRIES) {
                    throw new RuntimeException("Max retries exceeded", e);
                }

                logger.warn("Attempt {} failed, retrying in {}ms", attempt, b
ackoff);

                try {
                    Thread.sleep(backoff);
                } catch (InterruptedException ie) {
                    Thread.currentThread().interrupt();
                    throw new RuntimeException("Retry interrupted", ie);
                }

                backoff *= 2; // Exponential backoff
            }
        }
    }
}
```

Storage Solutions

Overview

Extensions can persist data using PostgreSQL with JSONB for flexible schema storage.

Database Access Pattern

```
@Inject
private DataSource dataSource;
```

```
public void saveData(String key, Map<String, Object> data) {
    String sql = """
        INSERT INTO extension_data (key, data, updated_at)
        VALUES (?, ?::jsonb, NOW())
        ON CONFLICT (key)
        DO UPDATE SET data = ?::jsonb, updated_at = NOW()
        """;

    try (Connection conn = dataSource.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {

        String jsonData = objectMapper.writeValueAsString(data);

        stmt.setString(1, key);
        stmt.setString(2, jsonData);
        stmt.setString(3, jsonData);

        stmt.executeUpdate();

    } catch (Exception e) {
        throw new RuntimeException("Failed to save data", e);
    }
}

public Map<String, Object> loadData(String key) {
    String sql = "SELECT data FROM extension_data WHERE key = ?";

    try (Connection conn = dataSource.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setString(1, key);

        try (ResultSet rs = stmt.executeQuery()) {
            if (rs.next()) {
                String jsonData = rs.getString("data");
                return objectMapper.readValue(jsonData, Map.class);
            }
            return null;
        }

    } catch (Exception e) {
        throw new RuntimeException("Failed to load data", e);
    }
}
```

Part VI: Quality and Operations

Testing Strategy

Overview

Comprehensive testing ensures your extension works correctly and handles edge cases gracefully.

Unit Testing

Test individual components in isolation:

```
public class UserTransformerTest {

    private UserTransformer transformer;

    @Before
    public void setUp() {
        transformer = new UserTransformer();
    }

    @Test
    public void testTransformValidUser() {
        ExternalUser externalUser = new ExternalUser();
        externalUser.setId("123");
        externalUser.setUsername("john.doe");
        externalUser.setEmail("john@example.com");
        externalUser.setFirstName("John");
        externalUser.setLastName("Doe");
        externalUser.setStatus("active");

        KristaUser result = transformer.transform(externalUser);

        assertNotNull(result);
        assertEquals("123", result.id());
        assertEquals("john.doe", result.username());
        assertEquals("john@example.com", result.email());
        assertEquals("John Doe", result.fullName());
        assertTrue(result.isActive());
    }

    @Test
    public void testTransformNullUser() {
        KristaUser result = transformer.transform(null);
        assertNull(result);
    }

    @Test
```

```
    public void testTransformInactiveUser() {
        ExternalUser externalUser = new ExternalUser();
        externalUser.setId("456");
        externalUser.setStatus("inactive");

        KristaUser result = transformer.transform(externalUser);

        assertFalse(result.isActive());
    }
}
```

Integration Testing

Test extension integration with external systems:

```
public class ApiIntegrationTest {

    private RestApiIntegration apiClient;

    @Before
    public void setUp() {
        // Use test environment
        apiClient = new RestApiIntegration(
            "https://api-test.example.com",
            "test-api-key"
        );
    }

    @Test
    public void testGetUsers() {
        List<User> users = apiClient.get("/users", UserList.class);

        assertNotNull(users);
        assertFalse(users.isEmpty());
    }

    @Test
    public void testCreateUser() {
        User newUser = new User("test@example.com", "Test User");

        User created = apiClient.post("/users", newUser, User.class);

        assertNotNull(created);
        assertNotNull(created.getId());
        assertEquals("test@example.com", created.getEmail());
    }

    @Test(expected = ApiException.class)
    public void testInvalidApiKey() {
        RestApiIntegration invalidClient = new RestApiIntegration(
```

```
            "https://api-test.example.com",
            "invalid-key"
        );

        invalidClient.get("/users", UserList.class);
    }
}
```

Validation Testing

Test attribute validation logic:

```
public class AttributeValidationTest {

    private MyExtension extension;

    @Before
    public void setUp() {
        extension = new MyExtension();
    }

    @Test
    public void testValidAttributes() {
        Map<String, Object> attributes = Map.of(
            "API Key", "valid-api-key-with-32-characters",
            "Host URL", "https://api.example.com",
            "Port", 443.0,
            "Use SSL", true
        );

        List<String> errors = extension.validateAttributes(attributes);

        assertTrue(errors.isEmpty());
    }

    @Test
    public void testMissingRequiredField() {
        Map<String, Object> attributes = Map.of(
            "Host URL", "https://api.example.com"
        );

        List<String> errors = extension.validateAttributes(attributes);

        assertFalse(errors.isEmpty());
        assertTrue(errors.stream().anyMatch(e -> e.contains("API Key is requi
red")));
    }

    @Test
    public void testInvalidUrl() {
```

```
        Map<String, Object> attributes = Map.of(
            "API Key", "valid-api-key-with-32-characters",
            "Host URL", "not-a-valid-url"
        );

        List<String> errors = extension.validateAttributes(attributes);

        assertFalse(errors.isEmpty());
        assertTrue(errors.stream().anyMatch(e -> e.contains("valid URL")));
    }
}
```

Testing Best Practices

1. **Test happy path and edge cases**: Cover both success and failure scenarios
2. **Use test data builders**: Create reusable test data factories
3. **Mock external dependencies**: Use Mockito for external API calls
4. **Test validation thoroughly**: Validate all attribute combinations
5. **Use descriptive test names**: Make test purpose clear
6. **Clean up test data**: Ensure tests don't leave artifacts

---

Error Handling and Best Practices

Overview

Robust error handling prevents failures and provides meaningful feedback to users and developers.

Error Handling Patterns

Pattern 1: Graceful Degradation

```
@CatalogRequest(CatalogRequest.Type.QUERY_SYSTEM)
public List<Entity> queryData(Map<String, Object> parameters) {
    try {
        return apiClient.fetchData(parameters);

    } catch (ApiException e) {
        logger.error("API call failed, returning cached data", e);
        return getCachedData();

    } catch (Exception e) {
        logger.error("Unexpected error during query", e);
        throw new RuntimeException("Failed to query data: " + e.getMessage(),
e);
    }
}
```

Pattern 2: Retry with Circuit Breaker

```
public class CircuitBreakerClient {

    private volatile CircuitState state = CircuitState.CLOSED;
    private int failureCount = 0;
    private static final int FAILURE_THRESHOLD = 5;

    public <T> T execute(Supplier<T> operation) {
        if (state == CircuitState.OPEN) {
            throw new CircuitBreakerOpenException("Circuit breaker is open");
        }

        try {
            T result = operation.get();
            onSuccess();
            return result;

        } catch (Exception e) {
            onFailure();
            throw e;
        }
    }

    private void onSuccess() {
        failureCount = 0;
        state = CircuitState.CLOSED;
    }

    private void onFailure() {
        failureCount++;
        if (failureCount >= FAILURE_THRESHOLD) {
            state = CircuitState.OPEN;
            logger.error("Circuit breaker opened after {} failures", failureC
ount);
        }
    }

    private enum CircuitState {
        CLOSED, OPEN
    }
}
```

Pattern 3: Detailed Error Messages

```
public List<String> validateAttributes(Map<String, Object> attributes) {
    List<String> errors = new ArrayList<>();

    // Provide specific, actionable error messages
    String apiKey = (String) attributes.get("API Key");
```

```
    if (apiKey == null || apiKey.isBlank()) {
        errors.add("API Key is required. Please enter your API key from the p
rovider dashboard.");
    } else if (apiKey.length() < 32) {
        errors.add("API Key must be at least 32 characters. Current length: "
+ apiKey.length());
    } else if (!apiKey.matches("^[A-Za-z0-9_-]+$")) {
        errors.add("API Key contains invalid characters. Only alphanumeric, u
nderscore, and hyphen are allowed.");
    }

    return errors;
}
```

Logging Best Practices

```
public class LoggingExample {

    private static final Logger logger = LoggerFactory.getLogger(LoggingExamp
le.class);

    public void processData(String userId, Map<String, Object> data) {
        // Log entry with context
        logger.info("Processing data for user: {}", userId);

        try {
            // Log important steps
            logger.debug("Validating data: {}", data.keySet());
            validateData(data);

            logger.debug("Transforming data");
            Map<String, Object> transformed = transformData(data);

            logger.debug("Saving data");
            saveData(userId, transformed);

            // Log success
            logger.info("Successfully processed data for user: {}", userId);

        } catch (ValidationException e) {
            // Log validation errors with details
            logger.warn("Validation failed for user {}: {}", userId, e.getMes
sage());
            throw e;

        } catch (Exception e) {
            // Log unexpected errors with full stack trace
            logger.error("Unexpected error processing data for user: " + user
Id, e);
            throw new RuntimeException("Failed to process data", e);
```

```
        }
    }
}
```

Best Practices Summary

1. **Always validate input**: Check for null, empty, and invalid values
2. **Provide meaningful errors**: Help users understand and fix problems
3. **Log at appropriate levels**: DEBUG for details, INFO for milestones, WARN for recoverable errors, ERROR for failures
4. **Include context in logs**: User IDs, request IDs, timestamps
5. **Handle exceptions gracefully**: Don't let exceptions propagate without context
6. **Use try-with-resources**: Ensure resources are cleaned up
7. **Fail fast**: Validate early and return errors immediately
8. **Document error codes**: Provide error code reference for users

---

Observability with OpenTelemetry

Overview

OpenTelemetry provides distributed tracing, metrics, and logging for monitoring extension performance and debugging issues.

Tracing Example

```
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.Tracer;

public class TracedExtension {

    private final Tracer tracer;

    @Inject
    public TracedExtension(Tracer tracer) {
        this.tracer = tracer;
    }

    @CatalogRequest(CatalogRequest.Type.QUERY_SYSTEM)
    public List<Entity> queryData(Map<String, Object> parameters) {
        Span span = tracer.spanBuilder("queryData")
            .setAttribute("operation", "query")
            .setAttribute("entityType", (String) parameters.get("entityType")
)
            .startSpan();

        try {
```

```
            // Your query logic
            List<Entity> results = performQuery(parameters);

            span.setAttribute("resultCount", results.size());
            span.setStatus(StatusCode.OK);

            return results;

        } catch (Exception e) {
            span.recordException(e);
            span.setStatus(StatusCode.ERROR, e.getMessage());
            throw e;

        } finally {
            span.end();
        }
    }
}
```

Metrics Example

```
import io.opentelemetry.api.metrics.LongCounter;
import io.opentelemetry.api.metrics.Meter;

public class MetricsExtension {

    private final LongCounter requestCounter;
    private final LongCounter errorCounter;

    @Inject
    public MetricsExtension(Meter meter) {
        this.requestCounter = meter.counterBuilder("extension.requests")
            .setDescription("Total number of requests")
            .build();

        this.errorCounter = meter.counterBuilder("extension.errors")
            .setDescription("Total number of errors")
            .build();
    }

    @CatalogRequest(CatalogRequest.Type.QUERY_SYSTEM)
    public List<Entity> queryData(Map<String, Object> parameters) {
        requestCounter.add(1);

        try {
            return performQuery(parameters);

        } catch (Exception e) {
            errorCounter.add(1);
            throw e;
```

```
        }
    }
}
```

Testing and Deployment

Deployment Checklist

Before deploying your extension:

- ☐ All unit tests pass
- ☐ Integration tests pass
- ☐ Validation logic tested thoroughly
- ☐ Error handling tested
- ☐ Logging configured correctly
- ☐ Secured fields encrypted
- ☐ Dependencies declared correctly
- ☐ Documentation updated
- ☐ Performance tested under load
- ☐ Security review completed

Deployment Process

1. **Build extension JAR**
   `gradle clean build`

2. **Verify JAR contents**
   `jar tf build/libs/my-extension-1.0.0.jar`

3. **Deploy to Krista platform**
   - Upload JAR through Krista admin interface
   - Configure extension attributes
   - Test connection
   - Deploy to production

Troubleshooting

Common Issues and Solutions

Issue 1: Extension not loading - **Symptom**: Extension doesn't appear in platform - **Causes**: Missing @Extension annotation, invalid domain ID, compilation errors - **Solution**: Check logs for errors, verify annotations, rebuild JAR

Issue 2: Dependency injection fails -
Symptom: NullPointerException when accessing injected dependencies
Causes: Missing @Inject annotation, wrong dependency name, circular dependencies
Solution: Verify @Named matches @Dependency name, check for circular dependencies

Issue 3: Attributes not saving -
Symptom: Configuration changes don't persist -
Causes: Validation errors, database connection issues, encryption failures -
Solution: Check validation logic, verify database connectivity, check logs for encryption errors

Issue 4: Authentication fails
Symptom: Users cannot authenticate
Causes: Invalid credentials, authenticator not registered, session timeout
Solution: Verify RequestAuthenticator implementation, check @InvokerRequest(AUTHENTICATOR) annotation

Issue 5: SSL/TLS connection fails
Symptom: Cannot connect to external service
Causes: Invalid certificate, wrong trust store, hostname mismatch
Solution: Verify certificate format (PKCS12), check trust store configuration, enable hostname verification

Debugging Tips

1. **Enable debug logging**: Set log level to DEBUG for detailed output
2. **Check platform logs**: Review Krista platform logs for errors
3. **Use breakpoints**: Debug extension code in IDE
4. **Test in isolation**: Test components independently
5. **Verify configuration**: Double-check all attribute values
6. **Check network connectivity**: Ensure external services are reachable

---

Part VII: Implementation Patterns and Integration

Implementation Patterns

Pattern: Modular Extension Architecture

Organize your extension into clear modules:

```
my-extension/
├── src/main/java/
│   ├── integration/          # External API integration (no Krista dependenc
ies)
```

```
│   │   ├── ApiClient.java
│   │   └── ExternalModels.java
│   ├── transformation/       # Data transformation layer
│   │   ├── UserTransformer.java
│   │   └── OrderTransformer.java
│   ├── controller/           # Krista request handlers
│   │   ├── UserArea.java
│   │   └── OrderArea.java
│   ├── entity/               # Krista entity definitions
│   │   ├── KristaUser.java
│   │   └── KristaOrder.java
│   └── config/               # Extension configuration
│       └── MyExtension.java
```

Pattern: Configuration Management

Centralize configuration access:

```java
public class ExtensionConfig {

    private final Map<String, Object> attributes;

    public ExtensionConfig(Map<String, Object> attributes) {
        this.attributes = new HashMap<>(attributes);
    }

    public String getApiUrl() {
        return getString("API URL");
    }

    public String getApiKey() {
        return getString("API Key");
    }

    public int getPort() {
        return getNumber("Port").intValue();
    }

    public boolean isSSLEnabled() {
        return getBoolean("Use SSL");
    }

    private String getString(String key) {
        Object value = attributes.get(key);
        return value != null ? value.toString() : null;
    }

    private Double getNumber(String key) {
        return (Double) attributes.get(key);
    }
```

```
    private Boolean getBoolean(String key) {
        return (Boolean) attributes.get(key);
    }
}
```

Integration with Krista Platform

Platform Integration Points

1. **Catalog Requests**: System integration operations
2. **Invoker Requests**: Lifecycle and interface handlers
3. **Entity Management**: Data model integration
4. **Authentication**: Security integration
5. **Storage**: Database integration
6. **Events**: Event-driven integration

Platform Services Available to Extensions

- **InvokerStore**: Configuration persistence
- **DataSource**: Database access
- **Tracer/Meter**: Observability
- **SessionManager**: Session management
- **EncryptionService**: Secure data encryption

Complete Extension Examples

Example 1: Simple REST API Extension

```
@Java(version = Java.Version.JAVA_21)
@Extension(
    implementingDomainIds = "catEntryDomain_12345678-1234-1234-1234-123456789
012",
    requireWorkspaceAdminRights = false
)
@Domain(
    id = "catEntryDomain_12345678-1234-1234-1234-123456789012",
    name = "Simple API",
    ecosystemId = "catEntryEcosystem_d3b05047-07b0-4b06-95a3-9fb8f7f608d9",
    ecosystemName = "Krista",
    ecosystemVersion = "41d20d96-9154-4f8e-a3ad-a435674fba5d"
)
@Field.Text(name = "API URL", required = true)
@Field.Text(name = "API Key", required = true, isSecured = true)
@Field.Number(name = "Timeout Seconds", required = false)
public final class SimpleApiExtension {
```

```
    private HttpClient httpClient;

    @InvokerRequest(InvokerRequest.Type.INVOKER_LOADED)
    public void onLoaded(Map<String, Object> attributes) {
        String apiUrl = (String) attributes.get("API URL");
        String apiKey = (String) attributes.get("API Key");
        Double timeout = (Double) attributes.getOrDefault("Timeout Seconds",
30.0);

        this.httpClient = HttpClient.newBuilder()
            .baseUrl(apiUrl)
            .header("Authorization", "Bearer " + apiKey)
            .connectTimeout(Duration.ofSeconds(timeout.longValue()))
            .build();
    }

    @InvokerRequest(InvokerRequest.Type.VALIDATE_ATTRIBUTES)
    public List<String> validateAttributes(Map<String, Object> attributes) {
        List<String> errors = new ArrayList<>();

        String apiUrl = (String) attributes.get("API URL");
        if (apiUrl == null || apiUrl.isBlank()) {
            errors.add("API URL is required");
        } else if (!isValidUrl(apiUrl)) {
            errors.add("API URL must be a valid URL");
        }

        String apiKey = (String) attributes.get("API Key");
        if (apiKey == null || apiKey.isBlank()) {
            errors.add("API Key is required");
        }

        return errors;
    }

    @CatalogRequest(CatalogRequest.Type.QUERY_SYSTEM)
    public List<Map<String, Object>> queryData(Map<String, Object> parameters
) {
        String endpoint = (String) parameters.get("endpoint");
        return httpClient.get(endpoint, List.class);
    }

    private boolean isValidUrl(String url) {
        try {
            new URL(url);
            return true;
        } catch (MalformedURLException e) {
            return false;
        }
```

```
    }
}
```

Conclusion

This guide has covered the complete extension development lifecycle for the Krista platform, from initial setup through production deployment. You now have the knowledge to:

- Design modular, maintainable extension architectures
- Implement Setup tab functionality with attributes, dependencies, and authentication
- Handle validation, trust configuration, and secure data storage
- Integrate with external systems using modern Java 21 patterns
- Test, debug, and deploy production-ready extensions
- Monitor and troubleshoot extensions in production

Next Steps

1. **Start with a simple extension**: Implement basic QUERY_SYSTEM operation
2. **Add Setup tab functionality**: Implement attributes and validation
3. **Add authentication**: Implement RequestAuthenticator
4. **Test thoroughly**: Write unit and integration tests
5. **Deploy and monitor**: Deploy to test environment and monitor performance
6. **Iterate and improve**: Gather feedback and enhance functionality

Additional Resources

- **kristagrand/extensions/**: Reference implementations
- **krista-service-apis-java/**: Extension APIs and KSDK components
- **Krista Platform Documentation**: Platform-specific guides
- **Java 21 Documentation**: Modern Java features and best practices

Guidelines for Creating Custom Agent Extensions

Custom agent extensions allow you to extend the Krista platform with your own specialized agents. Once correctly implemented, your extension will appear in the **Agent Setup → Agent List** section of the Krista interface, and its dashboard will be seamlessly embedded in the user's Home page under the Topic section.

To ensure full compatibility and successful registration, follow these mandatory technical specifications:

Hosting URL Endpoint

Your agent extension must expose a simple REST endpoint that tells the Krista platform where your agent's web-based dashboard is hosted. This is the only endpoint Krista will call automatically.

Requirements:

- **JAX-RS ID:** This must be set to default REST.
- **Endpoint Path:** /hostUrl.
- **Method**: GET
- **Response Structure:** The response must be a JSON object containing a hostingUrl string nested within a payload object. The hostingUrl should point to your deployed agent extension service.

Sample Implementation (Java)

```
@GET
@Path("/hostUrl")
public String getHostingUrl() {
    Map<String, Object> wrapper = new HashMap<>();
    Map<String, String> payload = new HashMap<>();

    // The hosting URL for your agent extension service
    payload.put("hostingUrl", "https://conversation-
agent.uslab.krista.app/");
    wrapper.put("payload", payload);

    return new Gson().toJson(wrapper);
}
```

Important Notes:

- The URL must be stable, publicly accessible, and support HTTPS.

- This endpoint should return quickly (ideally < 500ms) as it's called frequently by the platform.

Domain and Ecosystem Registration

Krista uses a catalog system to discover and register agent extensions. Your extension must be correctly associated with the official **Krista Ecosystem** and **Agent Domain**.

Required Values:

Field	Value
Ecosystem Name	Krista
Ecosystem ID	catEntryEcosystem_d3b05047-07b0-4b06-95a3-9fb8f7f608d9
Domain Name	Agent
Domain ID	catEntryDomain_c54d9930-ea03-4c58-bee6-26f90939171d
Ecosystem Version	9bc82a31-a79a-4dc8-bdbd-1ed7e576f7c1

Sample Domain Declaration

```
@Domain(
    id = "catEntryDomain_c54d9930-ea03-4c58-bee6-26f90939171d",
    name = "Agent",
    ecosystemId = "catEntryEcosystem_d3b05047-07b0-4b06-95a3-9fb8f7f608d9",
    ecosystemName = "Krista",
    ecosystemVersion = "9bc82a31-a79a-4dc8-bdbd-1ed7e576f7c1"
)
public class CustomAgentExtension {
    @CatalogRequest(
        id = "unique_request_id_here",
        name = "Extension Entry Point",
        description = "Description of the action",
        area = "Agent",
        type = CatalogRequest.Type.CHANGE_SYSTEM
    )
    public ExtensionResponse handleRequest() {
        // Implementation here
    }
}
```

Best Practices:

- Use a UUID or a clear, unique string for each `@CatalogRequest` ID.

- Provide meaningful, user-friendly names and detailed descriptions.
- Use appropriate request types (e.g., CHANGE_SYSTEM, LOOKUP, ACTION).

Technical Requirements & Common Pitfalls

Requirement	Details
Unique Identifiers	Every `@Domain` and `@CatalogRequest` must have a unique ID. Reuse will cause registration failure.
Entry Points	Each `@CatalogRequest` method is an entry point (action, lookup, system change, etc.).
Unsupported Methods	If a method in the required interface is not supported, throw `UnsupportedOperationException`.
Dashboard Rendering	Once registered, the dashboard loads via iframe from your `hostingUrl` in the Topic section.

Packaging, Testing & Deployment

Packaging Format:

- Package your extension as a single file: **Agent-Domain.daz**
- This is a special archive format recognized by Krista.

Catalog Integration Process:

Stage	Action	Where to Import
Development & Testing	Import into **Private Catalog** for internal verification	Private Catalog
Production	Add to **Global Catalog** for all Krista users to discover	Global Catalog

Recommended Workflow:

1. Build and test locally with the `/hostUrl` endpoint working.
2. Package as `.daz` file.
3. Import into your Private Catalog for testing.
4. Verify the agent appears in Agent Setup → Agent List and the dashboard loads correctly.

5. Once validated, submit to Global Catalog for production availability.

By following these guidelines precisely, your custom agent extension will integrate smoothly with the Krista platform, appear in the correct location, and provide a seamless experience for end users.

----------**Happy Extension Development**----------