

# CSV Import Enhancements

## *Value Mapping and Transformations*

Version 1.2.3, 2026-01-29T19:30:21Z

# Table of Contents

|   |    |
|---|----|
| 1. Executive Summary .....                            | 1  |
| 2. Current State Analysis .....                       | 2  |
| 2.1. Existing Architecture .....                      | 2  |
| 2.2. Current Limitations .....                        | 2  |
| 3. Proposed Design .....                              | 4  |
| 3.1. Overview .....                                   | 4  |
| 3.2. Component Design .....                           | 4  |
| 3.2.1. 1. ImportProfile Entity .....                  | 5  |
| 3.2.2. 2. ColumnMapping Configuration .....           | 6  |
| 3.2.3. 3. Supporting Enums and Classes .....          | 8  |
| 3.2.4. 4. LookupConfig for Reference Resolution ..... | 9  |
| 3.2.5. 5. Header-Based Field Metadata .....           | 10 |
| 3.2.6. 6. Intent Configuration .....                  | 20 |
| 3.2.7. 7. GlobalTransformations .....                 | 21 |
| 3.2.8. 8. LookupService Interface .....               | 22 |
| 3.2.9. 9. PreValidationTransformer Interface .....    | 23 |
| 3.2.10. 10. ImportRowContext .....                    | 24 |
| 3.2.11. 11. TransformationException .....             | 25 |
| 3.3. Enhanced Cell Processors .....                   | 26 |
| 3.3.1. ValueMapProcessor .....                        | 26 |
| 3.3.2. RegexReplaceProcessor .....                    | 28 |
| 3.3.3. CaseTransformProcessor .....                   | 28 |
| 3.3.4. LookupProcessor .....                          | 30 |
| 3.4. Enhanced CSVImportHelper .....                   | 31 |
| 3.4.1. New Method Signatures .....                    | 31 |
| 3.4.2. CSVFormatOptions (Refactored) .....            | 33 |
| 3.4.3. Enhanced Processor Chain Building .....        | 33 |
| 3.4.4. PreValidationTransformer Integration .....     | 36 |
| 3.4.5. Intent Resolution .....                        | 37 |
| 3.4.6. Intent Execution .....                         | 38 |
| 3.5. REST API Enhancements .....                      | 39 |
| 3.5.1. Updated Endpoints .....                        | 39 |
| 3.5.2. ImportProfile Resource .....                   | 40 |
| 3.6. Example Usage .....                              | 40 |
| 3.6.1. Example ImportProfile (JSON) .....             | 40 |
| 3.6.2. Example CSV (without intent column) .....      | 42 |
| 3.6.3. Example CSV (with intent column) .....         | 42 |
| 3.6.4. API Calls .....                                | 43 |

|   |    |
|---|----|
| 3.6.5. Example PreValidationTransformer .....                   | 43 |
| 4. Migration Path .....   | 46 |
| 4.1. Phase 1: Core Models (Non-Breaking) .....                  | 46 |
| 4.2. Phase 2: Cell Processors (Non-Breaking) .....              | 46 |
| 4.3. Phase 3: CSVImportHelper Enhancement (Non-Breaking) .....  | 46 |
| 4.4. Phase 4: PreValidationTransformer SPI (Non-Breaking) ..... | 46 |
| 4.5. Phase 5: Header Modifiers (Non-Breaking) .....             | 47 |
| 4.6. Phase 6: REST API Updates (Non-Breaking) .....             | 47 |
| 5. Security Considerations .....                                | 48 |
| 5.1. Profile Access Control .....                               | 48 |
| 5.2. Lookup Security .....                                      | 48 |
| 5.3. Regex Safety .....   | 48 |
| 6. Performance Considerations .....                             | 49 |
| 6.1. Lookup Caching .....                                       | 49 |
| 6.2. Batch Lookup Optimization .....                            | 49 |
| 6.3. Profile Caching .....                                      | 49 |
| 7. Testing Strategy .....                                       | 50 |
| 7.1. Unit Tests .....   | 50 |
| 7.2. Integration Tests .....                                    | 50 |
| 7.3. Performance Tests .....                                    | 50 |
| 8. Appendix A: Common Value Mapping Patterns .....              | 51 |
| 8.1. Boolean Values .....                                       | 51 |
| 8.2. Status Codes .....   | 51 |
| 8.3. Country Codes .....  | 51 |
| 9. Appendix B: Regex Patterns .....                             | 53 |
| 9.1. Remove Currency Symbols .....                              | 53 |
| 9.2. Normalize Phone Numbers .....                              | 53 |
| 9.3. Extract Numeric ID .....                                   | 53 |
| 9.4. Normalize Whitespace .....                                 | 53 |
| 10. Implementation Status .....                                 | 54 |
| 10.1. Completed Components .....                                | 54 |
| 10.2. REST API Endpoints .....                                  | 55 |
| 10.3. Usage .....   | 55 |
| 11. RowValueResolver: Per-Row Arbitrary Code Execution .....    | 57 |
| 11.1. Use Cases .....   | 57 |
| 11.2. Example: Address Geocoding .....                          | 57 |
| 11.2.1. 1. Create the RowValueResolver Implementation .....     | 57 |
| 11.2.2. 2. Configure the ImportProfile .....                    | 58 |
| 11.2.3. 3. Sample CSV Input .....                               | 59 |
| 11.2.4. 4. Result .....   | 59 |
| 11.3. RowValueResolver API .....                                | 60 |

11.4. ResolveResult Options ..... 60

11.5. Processing Order ..... 61

# Chapter 1. Executive Summary

This document proposes enhancements to the Quantum framework's CSV import functionality to add value mapping and transformation capabilities. The primary gaps addressed are:

1. **No value mapping** - Cannot map CSV values to different values prior to validation (e.g., "Y" → `true`, "Active" → `ACTIVE`)
2. **No lookup/reference resolution** - Cannot resolve foreign references by looking up values in other collections
3. **No pre-validation transformation hook** - Cannot programmatically transform parsed beans before validation
4. **No explicit intent control** - Cannot specify INSERT vs UPDATE per row; always uses auto-detect (UPSERT)

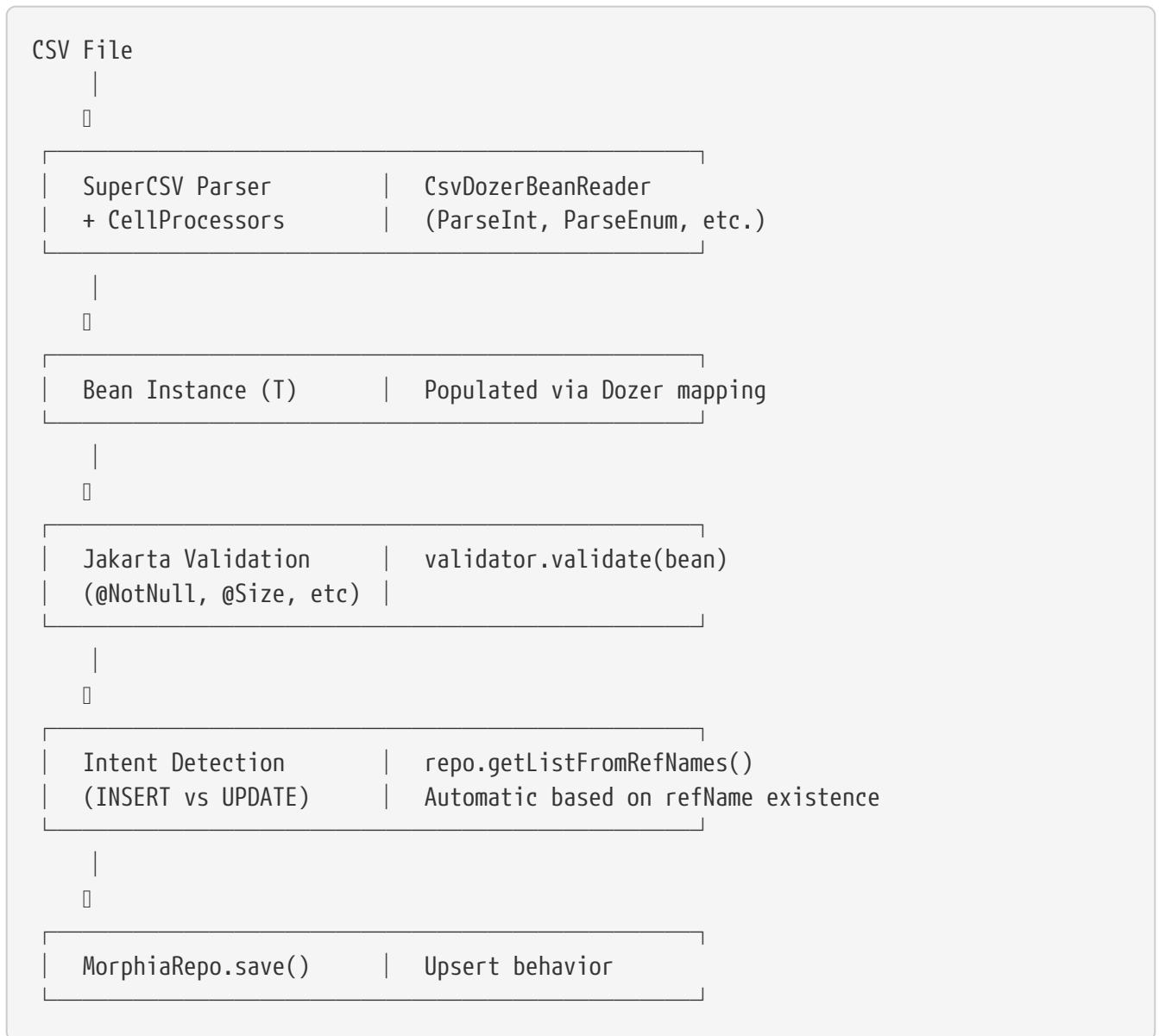


This design intentionally **excludes** MERGE and DELETE intents for safety reasons. Only INSERT, UPDATE, UPSERT, and SKIP are supported.

1. **No header-based field metadata** - Cannot specify required/optional/calculated fields via CSV header conventions

# Chapter 2. Current State Analysis

## 2.1. Existing Architecture



## 2.2. Current Limitations

| Gap                    | Current Behavior                                   | Desired Behavior   |
|------------------------|--|--|
| Value Mapping          | Only type conversion (String → int, String → enum) | Arbitrary value transformations via configurable mappings    |
| Lookup Resolution      | Not available                                      | Resolve references by looking up values in other collections |
| String Transformations | None   | Trim, case conversion, regex replacement                     |
| Pre-validation Hook    | None   | Transform bean after parsing, before validation              |

| Gap                         | Current Behavior                             | Desired Behavior   |
|-----------------------------|--|--|
| Default Values              | None (null if empty)                         | Configurable default values for empty/null cells                         |
| Intent Control              | Always UPSERT (auto-detect INSERT vs UPDATE) | Optional per-row intent column (INSERT, UPDATE, SKIP) with validation    |
| Header-based Field Metadata | None - all fields treated equally            | Suffix conventions for required (*), optional (?), calculated (~) fields |

# Chapter 3. Proposed Design

## 3.1. Overview

The design introduces three new concepts:

1. **ImportProfile** - Configurable transformation and mapping rules
2. **Enhanced CellProcessors** - Value mapping, lookup, and string transformation processors
3. **PreValidationTransformer** - Programmatic transformation hook

CSV File + ImportProfile

|

□

SuperCSV Parser

+ Enhanced Processors

|

← ValueMapProcessor, LookupProcessor

|

□

Bean Instance (T)

|

|

□

PreValidationTransform

|

← NEW: Programmatic transformation

|

□

Jakarta Validation

|

|

□

Intent Detection

|

(unchanged)

|

□

MorphiaRepo.save()

|

(unchanged)

## 3.2. Component Design



### 3.2.1. 1. ImportProfile Entity

A persistent, reusable configuration for import transformations.

```
package com.e2eq.framework.model.persistent.imports;

import com.e2eq.framework.model.persistent.base.BaseModel;
import dev.morphia.annotations.Entity;
import io.quarkus.runtime.annotations.RegisterForReflection;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import lombok.experimental.SuperBuilder;

import java.util.List;

/**
 * Persistent configuration for CSV import transformations.
 * Profiles can be reused across multiple imports and shared within a tenant.
 */
@Data
@EqualsAndHashCode(callSuper = true)
@RegisterForReflection
@SuperBuilder
@NoArgsConstructor
@Entity
public class ImportProfile extends BaseModel {

    /**
     * Target entity class name (fully qualified).
     * If null, profile can be used with any entity type.
     */
    private String targetType;

    /**
     * Column-to-field mappings with transformations.
     * Order matters - defines CSV column order when requestedColumns not specified.
     */
    private List<ColumnMapping> columnMappings;

    /**
     * Global transformations applied to all string fields.
     */
    private GlobalTransformations globalTransformations;

    /**
     * Whether to fail on first transformation error or collect all errors.
     */
    private boolean failFast = false;
```

```

/**
 * Optional description of what this profile is for.
 */
private String description;

/**
 * Optional column name containing per-row intent.
 * If specified, the CSV can include a column with values: INSERT, UPDATE, SKIP.
 * If null, intent is determined automatically (UPSERT behavior).
 *
 * Note: Only INSERT, UPDATE, and SKIP are supported. MERGE and DELETE
 * are intentionally not supported for safety reasons.
 */
private String intentColumn;

/**
 * Default intent when intentColumn is specified but a row has empty/invalid
value.
 * One of: INSERT, UPDATE, UPSERT (auto-detect), SKIP.
 * Default: UPSERT (auto-detect based on refName existence).
 */
private String defaultIntent = "UPSERT";

@Override
public String bmFunctionalArea() {
    return "IMPORTS";
}

@Override
public String bmFunctionalDomain() {
    return "IMPORTS";
}
}

```

### 3.2.2. 2. ColumnMapping Configuration

```

package com.e2eq.framework.model.persistent.imports;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;
import lombok.Builder;

import java.util.Map;

/**
 * Configuration for mapping and transforming a single CSV column.
 */
@Data

```

```

@NoArgsConstructor
@AllArgsConstructor
@Builder
public class ColumnMapping {

    /**
     * CSV column index (0-based) or column name if header present.
     */
    private String sourceColumn;

    /**
     * Target field name on the entity.
     * Supports nested paths (e.g., "address.city") and array notation (e.g.,
"tags[0]").
     */
    private String targetField;

    /**
     * Static value mappings: CSV value → target value.
     * Applied before type conversion.
     * Example: {"Y": "true", "N": "false"}
     */
    private Map<String, String> valueMappings;

    /**
     * Whether value mapping comparison is case-sensitive.
     * Default: false (case-insensitive)
     */
    @Builder.Default
    private boolean valueMappingCaseSensitive = false;

    /**
     * Behavior when CSV value not found in valueMappings.
     * Default: PASSTHROUGH
     */
    @Builder.Default
    private UnmappedValueBehavior unmappedValueBehavior = UnmappedValueBehavior
.PASSTHROUGH;

    /**
     * Regex pattern for replacement.
     */
    private String regexPattern;

    /**
     * Replacement string for regex matches.
     * Supports capture group references ($1, $2, etc.)
     */
    private String regexReplacement;

    /**

```

```

    * Default value if CSV cell is empty/null.
    * Applied after all other transformations.
    */
    private String defaultValue;

    /**
     * Trim leading/trailing whitespace.
     * Default: true
     */
    @Builder.Default
    private boolean trim = true;

    /**
     * Convert empty string to null.
     * Default: true
     */
    @Builder.Default
    private boolean emptyToNull = true;

    /**
     * Case transformation: UPPER, LOWER, TITLE, NONE.
     * Default: NONE
     */
    @Builder.Default
    private CaseTransform caseTransform = CaseTransform.NONE;

    /**
     * Lookup configuration for resolving references from other collections.
     */
    private LookupConfig lookup;

    /**
     * Date/time format pattern for parsing date strings.
     * Example: "yyyy-MM-dd", "MM/dd/yyyy HH:mm:ss"
     */
    private String dateFormat;

    /**
     * Locale for date/number parsing (e.g., "en-US", "de-DE").
     */
    private String locale;
}

```

### 3.2.3. 3. Supporting Enums and Classes

```

package com.e2eq.framework.model.persistent.imports;

/**
 * Behavior when a CSV value is not found in the value mappings.

```

```

*/
public enum UnmappedValueBehavior {
    /** Keep the original CSV value unchanged */
    PASSTHROUGH,
    /** Convert to null */
    NULL,
    /** Treat as an error */
    FAIL
}

/**
 * String case transformation options.
 */
public enum CaseTransform {
    NONE,
    UPPER,
    LOWER,
    TITLE
}

```

### 3.2.4. 4. LookupConfig for Reference Resolution

Supports looking up values from other collections (e.g., map customer name to customer ID).

```

package com.e2eq.framework.model.persistent.imports;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;
import lombok.Builder;

/**
 * Configuration for looking up values from another collection.
 * Useful for resolving foreign key references during import.
 *
 * Example: Map customer name from CSV to customer refName in database.
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class LookupConfig {

    /**
     * Collection/entity class name to lookup from.
     * Can be simple name (e.g., "Customer") or fully qualified.
     */
    private String lookupCollection;

    /**

```

```

    * Field in lookup collection to match against CSV value.
    * Example: "displayName", "email", "externalId"
    */
    private String lookupMatchField;

    /**
     * Field in lookup collection to return as the mapped value.
     * Example: "refName", "id"
     */
    private String lookupReturnField;

    /**
     * Behavior when lookup fails to find a match.
     * Default: FAIL
     */
    @Builder.Default
    private LookupFailBehavior onNotFound = LookupFailBehavior.FAIL;

    /**
     * Whether to cache lookup results for performance.
     * Recommended for large imports with repeated values.
     * Default: true
     */
    @Builder.Default
    private boolean cacheLookups = true;

    /**
     * Optional filter to apply to lookup query.
     * Uses standard Quantum query syntax.
     * Example: "status:ACTIVE"
     */
    private String lookupFilter;
}

/**
 * Behavior when lookup fails to find a matching record.
 */
public enum LookupFailBehavior {
    /** Mark row as error */
    FAIL,
    /** Set field to null */
    NULL,
    /** Keep original CSV value unchanged */
    PASSTHROUGH
}

```

### 3.2.5. 5. Header-Based Field Metadata

CSV headers can include optional suffix modifiers to specify field behavior. This feature is **opt-in** via the `enableHeaderModifiers` flag.

## Header Modifier Syntax

```
fieldName[modifier]
```

Where modifier is one of:

- \* = Required (fail if empty/null)
- ? = Optional (allow empty/null, skip validation)
- ~ = Calculated (compute if not provided)
- # = Key field (used for intent detection instead of refName)

## Examples

```
refName*,displayName*,description?,price*,sku#,createdAt~,modifiedAt~
SKU-001,Widget,A great widget,19.99,SKU-001,,
SKU-002,Gadget,,29.99,SKU-002,,
```

In this example: - **refName\*** and **displayName\*** are required - import fails if empty - **description?** is optional - empty values are accepted without validation error - **price\*** is required - **sku#** is the key field - used instead of **refName** for INSERT vs UPDATE detection - **createdAt~** and **modifiedAt~** are calculated - if empty, system generates values

## HeaderModifier Enum

```
package com.e2eq.framework.util.csv;

/**
 * Modifiers that can be applied to CSV header columns via suffix notation.
 * Feature is opt-in via ImportProfile.enableHeaderModifiers or REST parameter.
 */
public enum HeaderModifier {
    /**
     * No modifier - field uses default behavior from entity validation.
     */
    NONE(""),

    /**
     * Required field - import fails if value is empty/null.
     * Checked BEFORE Jakarta validation, providing earlier feedback.
     * Suffix: *
     */
    REQUIRED("*"),

    /**
     * Optional field - empty/null values are allowed.
     * Skips @NotNull/@NotEmpty validation for this field.
     * Suffix: ?
     */
    OPTIONAL("?"),
```

```

/**
 * Calculated field - if empty, value is computed by a FieldCalculator.
 * Common uses: timestamps, UUIDs, derived values.
 * Suffix: ~
 */
CALCULATED("~"),

/**
 * Key field - used for INSERT vs UPDATE detection instead of refName.
 * Useful when importing data where refName differs from natural key.
 * Only one key field allowed per import.
 * Suffix: #
 */
KEY("#");

private final String suffix;

HeaderModifier(String suffix) {
    this.suffix = suffix;
}

public String getSuffix() {
    return suffix;
}

/**
 * Parse modifier from header column name.
 * @param header The raw header (e.g., "displayName*", "price?")
 * @return The modifier, or NONE if no recognized suffix
 */
public static HeaderModifier fromHeader(String header) {
    if (header == null || header.isEmpty()) return NONE;
    char last = header.charAt(header.length() - 1);
    return switch (last) {
        case '*' -> REQUIRED;
        case '?' -> OPTIONAL;
        case '~' -> CALCULATED;
        case '#' -> KEY;
        default -> NONE;
    };
}

/**
 * Strip modifier suffix from header to get field name.
 * @param header The raw header (e.g., "displayName*")
 * @return The field name without suffix (e.g., "displayName")
 */
public static String stripModifier(String header) {
    if (header == null || header.isEmpty()) return header;
    char last = header.charAt(header.length() - 1);

```



```

        if (last == '*' || last == '?' || last == '~' || last == '#') {
            return header.substring(0, header.length() - 1);
        }
        return header;
    }
}

```

## ParsedHeader Class

```

package com.e2eq.framework.util.csv;

import lombok.Value;

/**
 * Represents a parsed CSV header with field name and modifier.
 */
@Value
public class ParsedHeader {
    String fieldName;
    HeaderModifier modifier;

    /**
     * Parse a raw header string into field name and modifier.
     */
    public static ParsedHeader parse(String rawHeader) {
        HeaderModifier modifier = HeaderModifier.fromHeader(rawHeader);
        String fieldName = HeaderModifier.stripModifier(rawHeader);
        return new ParsedHeader(fieldName, modifier);
    }

    /**
     * @return true if this field is marked as required (*)
     */
    public boolean isRequired() {
        return modifier == HeaderModifier.REQUIRED;
    }

    /**
     * @return true if this field is marked as optional
     */
    public boolean isOptional() {
        return modifier == HeaderModifier.OPTIONAL;
    }

    /**
     * @return true if this field should be calculated when empty
     */
    public boolean isCalculated() {
        return modifier == HeaderModifier.CALCULATED;
    }
}

```

```

/**
 * @return true if this field is the key for intent detection (#)
 */
public boolean isKey() {
    return modifier == HeaderModifier.KEY;
}
}

```

## FieldCalculator SPI

For calculated fields (~ modifier), a pluggable calculator interface:

```

package com.e2eq.framework.util.csv;

import com.e2eq.framework.model.persistent.base.UnversionedBaseModel;

/**
 * SPI for calculating field values when marked with ~ modifier and value is empty.
 * Implementations are discovered via CDI.
 */
public interface FieldCalculator {

    /**
     * @return The field name this calculator handles (e.g., "createdAt", "uuid")
     */
    String getFieldName();

    /**
     * @return Entity types this calculator applies to, or empty for all types
     */
    default Class<?>[] getTargetTypes() {
        return new Class<?>[0]; // All types
    }

    /**
     * @return Priority (lower = earlier). Default calculators run at 1000.
     */
    default int getPriority() {
        return 1000;
    }

    /**
     * Calculate the value for a field.
     *
     * @param bean The entity being imported
     * @param fieldName The field to calculate
     * @param context Import context
     * @return The calculated value
     */
}

```

```
    Object calculate(UnversionedBaseModel bean, String fieldName, ImportRowContext context);
}
```

## Built-in Field Calculators

```
/**
 * Calculates timestamp fields (createdAt, modifiedAt, updatedAt).
 */
@ApplicationScoped
public class TimestampFieldCalculator implements FieldCalculator {

    private static final Set<String> TIMESTAMP_FIELDS = Set.of(
        "createdAt", "modifiedAt", "updatedAt", "createdAt", "modifiedDate"
    );

    @Override
    public String getFieldName() {
        return null; // Handles multiple fields
    }

    public boolean handles(String fieldName) {
        return TIMESTAMP_FIELDS.contains(fieldName);
    }

    @Override
    public Object calculate(UnversionedBaseModel bean, String fieldName,
        ImportRowContext context) {
        return Instant.now();
    }
}

/**
 * Calculates UUID fields.
 */
@ApplicationScoped
public class UUIDFieldCalculator implements FieldCalculator {

    @Override
    public String getFieldName() {
        return "uuid";
    }

    @Override
    public Object calculate(UnversionedBaseModel bean, String fieldName,
        ImportRowContext context) {
        return UUID.randomUUID().toString();
    }
}
```

```

/**
 * Calculates refName from other fields if not provided.
 */
@ApplicationScoped
public class RefNameCalculator implements FieldCalculator {

    @Override
    public String getFieldName() {
        return "refName";
    }

    @Override
    public int getPriority() {
        return 500; // Run early
    }

    @Override
    public Object calculate(UnversionedBaseModel bean, String fieldName,
ImportRowContext context) {
        // Try to derive from key field or generate UUID
        if (bean.getDisplayName() != null && !bean.getDisplayName().isBlank()) {
            return slugify(bean.getDisplayName());
        }
        return UUID.randomUUID().toString();
    }

    private String slugify(String input) {
        return input.toLowerCase()
            .replaceAll("[^a-z0-9]+", "-")
            .replaceAll("^-|-$", "");
    }
}

```

## ImportProfile Configuration

Add to ImportProfile:

```

/**
 * Enable header modifier parsing (!, ?, ~, *).
 * When false (default), headers are treated as plain field names.
 * This maintains backward compatibility.
 */
private boolean enableHeaderModifiers = false;

/**
 * Custom field calculators defined in the profile.
 * These supplement CDI-discovered calculators.
 */
private List<InlineFieldCalculator> inlineCalculators;

```

```

/**
 * Inline field calculator defined within an ImportProfile.
 * Supports simple expression-based calculations.
 */
@Data
public class InlineFieldCalculator {
    /**
     * Field name to calculate.
     */
    private String fieldName;

    /**
     * Calculation type.
     */
    private CalculationType type;

    /**
     * Static value (for STATIC type).
     */
    private String staticValue;

    /**
     * Source field (for COPY type).
     */
    private String sourceField;

    /**
     * Template with ${fieldName} placeholders (for TEMPLATE type).
     */
    private String template;

    public enum CalculationType {
        /** Use current timestamp */
        TIMESTAMP,
        /** Generate UUID */
        UUID,
        /** Use static value */
        STATIC,
        /** Copy from another field */
        COPY,
        /** Template with field placeholders */
        TEMPLATE
    }
}

```

## REST API Parameter

```

@Parameter(description = "Enable header modifier parsing (*, ?, ~, #). " +
    "When true, headers like 'name*' are parsed as required

```

```
fields.")
@QueryParam("enableHeaderModifiers") @DefaultValue("false") boolean
enableHeaderModifiers
```

## Processing Logic

```
/**
 * Process headers and extract field metadata.
 */
private List<ParsedHeader> parseHeaders(
    String[] rawHeaders,
    boolean enableModifiers) {

    List<ParsedHeader> parsed = new ArrayList<>(rawHeaders.length);
    int keyFieldCount = 0;

    for (String raw : rawHeaders) {
        if (enableModifiers) {
            ParsedHeader header = ParsedHeader.parse(raw);
            if (header.isKey()) {
                keyFieldCount++;
                if (keyFieldCount > 1) {
                    throw new ImportException(
                        "Multiple key fields (#) specified. Only one key field is
allowed.");
                }
            }
            parsed.add(header);
        } else {
            // No modifier parsing - treat as plain field name
            parsed.add(new ParsedHeader(raw, HeaderModifier.NONE));
        }
    }

    return parsed;
}

/**
 * Validate required fields after parsing, before Jakarta validation.
 */
private void validateRequiredFields(
    Object bean,
    List<ParsedHeader> headers,
    Map<String, Object> values,
    ImportRowResult<?> rowResult) {

    for (ParsedHeader header : headers) {
        if (header.isRequired()) {
            Object value = values.get(header.getFieldName());
            if (value == null || (value instanceof String s && s.isBlank())) {
```

```

        rowResult.addError(new FieldError(
            header.getFieldName(),
            "Field is required (marked with *) but value is empty",
            FieldErrorCode.VALIDATION
        ));
    }
}

/**
 * Apply calculated fields.
 */
private void applyCalculatedFields(
    UnversionedBaseModel bean,
    List<ParsedHeader> headers,
    ImportRowContext context) {

    for (ParsedHeader header : headers) {
        if (header.isCalculated()) {
            Object currentValue = getFieldValue(bean, header.getFieldName());
            if (currentValue == null || (currentValue instanceof String s && s.
isBlank())) {
                Object calculated = calculateField(bean, header.getFieldName(),
context);
                if (calculated != null) {
                    setFieldValue(bean, header.getFieldName(), calculated);
                }
            }
        }
    }
}

/**
 * Find key field for intent detection.
 */
private String findKeyField(List<ParsedHeader> headers) {
    for (ParsedHeader header : headers) {
        if (header.isKey()) {
            return header.getFieldName();
        }
    }
    return "refName"; // Default
}

```

### Example: Complete CSV with All Modifiers

```

_action,sku#,displayName*,description?,price*,category*,createdAt~,uuid~
INSERT,NEW-001,New Widget,A great new widget,19.99,Electronics,,
UPDATE,EXIST-001,Updated Name,,29.99,Home,,

```

```
UPSERT,AUTO-001,Auto Detect,Optional desc,39.99,Tools,,
SKIP,IGNORE-001,Ignored Row,,,,,
```

### Header interpretation:

| Header                    | Modifier   | Behavior   |
|---------------------------|------------|--|
| <code>_action</code>      | none       | Intent column (configured separately)  |
| <code>sku#</code>         | KEY        | Use <code>sku</code> instead of <code>refName</code> for INSERT/UPDATE detection |
| <code>displayName*</code> | REQUIRED   | Fail import if empty   |
| <code>description?</code> | OPTIONAL   | Allow empty, skip @NotNull validation  |
| <code>price*</code>       | REQUIRED   | Fail import if empty   |
| <code>category*</code>    | REQUIRED   | Fail import if empty   |
| <code>createdAt~</code>   | CALCULATED | Generate timestamp if empty  |
| <code>uuid~</code>        | CALCULATED | Generate UUID if empty   |

### Backward Compatibility

- **Default behavior unchanged:** `enableHeaderModifiers=false` by default
- **Opt-in only:** Must explicitly enable via profile or REST parameter
- **Graceful handling:** If disabled, characters like `*?~#` in field names are treated literally
- **Escape mechanism:** To use literal `in a field name` when modifiers are enabled, use `\\`

### 3.2.6. 6. Intent Configuration

The existing `Intent` enum (INSERT, UPDATE, SKIP) can optionally be specified per-row via a designated CSV column.

```
package com.e2eq.framework.util.csv;

/**
 * Supported intent values for CSV import.
 * These match the existing CSVImportHelper.Intent enum.
 *
 * NOTE: MERGE and DELETE are intentionally NOT supported to prevent
 * accidental data loss or complex partial updates via CSV.
 */
public enum ImportIntent {
    /**
     * Insert a new record. Fails if refName already exists.
     */
    INSERT,

    /**
```



```

    * Update an existing record. Fails if refName does not exist.
    */
    UPDATE,

    /**
     * Auto-detect: INSERT if new, UPDATE if exists.
     * This is the current default behavior.
     */
    UPSERT,

    /**
     * Skip this row - do not process.
     * Useful for conditional imports or marking rows to ignore.
     */
    SKIP
}

```

### Why MERGE and DELETE are not supported:

- **DELETE:** Allowing bulk deletes via CSV is dangerous. A malformed file or user error could delete large amounts of data. Deletes should be explicit via the REST DELETE endpoint or purpose-built batch operations with additional safeguards.
- **MERGE:** Partial/merge updates are complex and error-prone. It's unclear which fields should be preserved vs. overwritten when the CSV value is empty. The current full-replacement semantic is predictable and explicit.

### 3.2.7. 7. GlobalTransformations

```

package com.e2eq.framework.model.persistent.imports;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;
import lombok.Builder;

/**
 * Global transformations applied to all string fields during import.
 * These are applied before column-specific transformations.
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class GlobalTransformations {

    /**
     * Trim leading/trailing whitespace from all string fields.
     * Default: true
     */
}

```

```

@Builder.Default
private boolean trimStrings = true;

/**
 * Convert empty strings to null.
 * Default: true
 */
@Builder.Default
private boolean emptyStringsToNull = true;

/**
 * Unicode normalization form: NFC, NFD, NFKC, NFKD, or null for none.
 */
private String unicodeNormalization;

/**
 * Remove ASCII control characters (0x00-0x1F except tab/newline).
 * Default: false
 */
@Builder.Default
private boolean removeControlChars = false;

/**
 * Maximum string length. Strings exceeding this are truncated.
 * Null means no limit.
 */
private Integer maxStringLength;

/**
 * Replace multiple consecutive whitespace with single space.
 * Default: false
 */
@Builder.Default
private boolean normalizeWhitespace = false;
}

```

### 3.2.8. 8. LookupService Interface

```

package com.e2eq.framework.util.csv;

import com.e2eq.framework.model.persistent.imports.LookupConfig;
import java.util.List;
import java.util.Map;
import java.util.Optional;

/**
 * Service for resolving lookup references during CSV import.
 */
public interface LookupService {

```

```

/**
 * Look up a single value.
 *
 * @param config The lookup configuration
 * @param matchValue The value to match against lookupMatchField
 * @return The resolved value, or empty if not found
 */
Optional<String> lookup(LookupConfig config, String matchValue);

/**
 * Batch lookup for performance optimization.
 * Implementations should use IN queries where possible.
 *
 * @param config The lookup configuration
 * @param matchValues The values to match
 * @return Map of matchValue → resolvedValue (missing keys = not found)
 */
Map<String, String> batchLookup(LookupConfig config, List<String> matchValues);

/**
 * Clear any cached lookup results.
 */
void clearCache();
}

```

### 3.2.9. 9. PreValidationTransformer Interface

Programmatic hook for complex transformations that can't be expressed declaratively.

```

package com.e2eq.framework.util.csv;

import com.e2eq.framework.model.persistent.base.UnversionedBaseModel;

/**
 * SPI for transforming parsed CSV beans before validation.
 * Implementations are discovered via CDI and applied in priority order.
 *
 * Use cases:
 * - Computing derived fields
 * - Cross-field validation/transformation
 * - Complex business logic that can't be expressed declaratively
 * - Conditional transformations based on other field values
 */
public interface PreValidationTransformer<T extends UnversionedBaseModel> {

    /**
     * @return The entity class this transformer handles
     */
    Class<T> getTargetType();
}

```

```

/**
 * @return Priority (lower = earlier). Default transformers run at 1000.
 */
default int getPriority() {
    return 1000;
}

/**
 * Transform the bean after CSV parsing but before validation.
 *
 * @param bean The parsed bean instance (already has cell processor
transformations applied)
 * @param context Import context with access to row data, profile, lookup service,
etc.
 * @return The transformed bean (may be same instance modified in place, or a new
instance)
 * @throws TransformationException if transformation fails (row will be marked as
error)
 */
T transform(T bean, ImportRowContext context) throws TransformationException;
}

```

### 3.2.10. 10. ImportRowContext

```

package com.e2eq.framework.util.csv;

import com.e2eq.framework.model.persistent.imports.ImportProfile;
import lombok.Builder;
import lombok.Data;

import java.util.List;
import java.util.Map;

/**
 * Context information available to PreValidationTransformer implementations.
 */
@Data
@Builder
public class ImportRowContext {

    /**
     * Current row number (1-based, after header).
     */
    private int rowNumber;

    /**
     * Raw CSV line as string.
     */
}

```

```

private String rawLine;

/**
 * Parsed cell values by column index (0-based).
 * These are the raw string values before any transformation.
 */
private List<String> rawValues;

/**
 * Parsed cell values by column name (if header was present).
 * These are the raw string values before any transformation.
 */
private Map<String, String> rawValuesByName;

/**
 * The import profile being used (may be null for profile-less imports).
 */
private ImportProfile profile;

/**
 * The import session ID (if using session-based flow).
 */
private String sessionId;

/**
 * Access to lookup service for additional reference resolution.
 */
private LookupService lookupService;
}

```

### 3.2.11. 11. TransformationException

```

package com.e2eq.framework.util.csv;

/**
 * Exception thrown when a transformation fails during CSV import.
 * Results in the row being marked as an error.
 */
public class TransformationException extends RuntimeException {

    private final String field;

    public TransformationException(String message) {
        super(message);
        this.field = null;
    }

    public TransformationException(String field, String message) {
        super(message);
    }
}

```

```

        this.field = field;
    }

    public TransformationException(String message, Throwable cause) {
        super(message, cause);
        this.field = null;
    }

    public TransformationException(String field, String message, Throwable cause) {
        super(message, cause);
        this.field = field;
    }

    /**
     * @return The field that caused the error, or null if not field-specific
     */
    public String getField() {
        return field;
    }
}

```

## 3.3. Enhanced Cell Processors

### 3.3.1. ValueMapProcessor

```

package com.e2eq.framework.util.csv.processors;

import com.e2eq.framework.model.persistent.imports.UnmappedValueBehavior;
import org.supercsv.cellprocessor.CellProcessorAdaptor;
import org.supercsv.cellprocessor.ift.CellProcessor;
import org.supercsv.exception.SuperCsvCellProcessorException;
import org.supercsv.util.CsvContext;

import java.util.HashMap;
import java.util.Map;

/**
 * Cell processor that maps values using a lookup table.
 */
public class ValueMapProcessor extends CellProcessorAdaptor {

    private final Map<String, String> mappings;
    private final Map<String, String> lowerCaseMappings;
    private final boolean caseSensitive;
    private final UnmappedValueBehavior unmappedBehavior;

    public ValueMapProcessor(
        Map<String, String> mappings,
        boolean caseSensitive,

```

```

        UnmappedValueBehavior unmappedBehavior,
        CellProcessor next) {
    super(next);
    this.mappings = mappings;
    this.caseSensitive = caseSensitive;
    this.unmappedBehavior = unmappedBehavior;

    // Pre-compute lowercase keys for case-insensitive matching
    if (!caseSensitive) {
        this.lowerCaseMappings = new HashMap<>();
        mappings.forEach((k, v) -> lowerCaseMappings.put(k.toLowerCase(), v));
    } else {
        this.lowerCaseMappings = null;
    }
}

@Override
public Object execute(Object value, CsvContext ctx) {
    if (value == null) {
        return next.execute(null, ctx);
    }

    String stringValue = value.toString();
    String mapped;

    if (caseSensitive) {
        mapped = mappings.get(stringValue);
    } else {
        mapped = lowerCaseMappings.get(stringValue.toLowerCase());
    }

    if (mapped == null) {
        switch (unmappedBehavior) {
            case PASSTHROUGH:
                return next.execute(value, ctx);
            case NULL:
                return next.execute(null, ctx);
            case FAIL:
                throw new SuperCsvCellProcessorException(
                    String.format("No mapping found for value '%s'. Valid values:
%s",
                                stringValue, mappings.keySet()),
                    ctx, this);
            default:
                return next.execute(value, ctx);
        }
    }

    return next.execute(mapped, ctx);
}

```

```
}
```

### 3.3.2. RegexReplaceProcessor

```
package com.e2eq.framework.util.csv.processors;

import org.supercsv.cellprocessor.CellProcessorAdaptor;
import org.supercsv.cellprocessor.ift.CellProcessor;
import org.supercsv.util.CsvContext;

import java.util.regex.Pattern;

/**
 * Cell processor that performs regex replacement on string values.
 */
public class RegexReplaceProcessor extends CellProcessorAdaptor {

    private final Pattern pattern;
    private final String replacement;

    public RegexReplaceProcessor(String regex, String replacement, CellProcessor next)
    {
        super(next);
        this.pattern = Pattern.compile(regex);
        this.replacement = replacement != null ? replacement : "";
    }

    @Override
    public Object execute(Object value, CsvContext ctx) {
        if (value == null) {
            return next.execute(null, ctx);
        }

        String result = pattern.matcher(value.toString()).replaceAll(replacement);
        return next.execute(result, ctx);
    }
}
```

### 3.3.3. CaseTransformProcessor

```
package com.e2eq.framework.util.csv.processors;

import com.e2eq.framework.model.persistent.imports.CaseTransform;
import org.supercsv.cellprocessor.CellProcessorAdaptor;
import org.supercsv.cellprocessor.ift.CellProcessor;
import org.supercsv.util.CsvContext;

/**
```



```

* Cell processor that transforms string case.
*/
public class CaseTransformProcessor extends CellProcessorAdaptor {

    private final CaseTransform transform;

    public CaseTransformProcessor(CaseTransform transform, CellProcessor next) {
        super(next);
        this.transform = transform;
    }

    @Override
    public Object execute(Object value, CsvContext ctx) {
        if (value == null) {
            return next.execute(null, ctx);
        }

        String s = value.toString();
        String result = switch (transform) {
            case UPPER -> s.toUpperCase();
            case LOWER -> s.toLowerCase();
            case TITLE -> toTitleCase(s);
            case NONE -> s;
        };

        return next.execute(result, ctx);
    }

    private String toTitleCase(String s) {
        if (s == null || s.isEmpty()) return s;

        StringBuilder result = new StringBuilder(s.length());
        boolean capitalizeNext = true;

        for (char c : s.toCharArray()) {
            if (Character.isWhitespace(c)) {
                capitalizeNext = true;
                result.append(c);
            } else if (capitalizeNext) {
                result.append(Character.toUpperCase(c));
                capitalizeNext = false;
            } else {
                result.append(Character.toLowerCase(c));
            }
        }

        return result.toString();
    }
}

```

### 3.3.4. LookupProcessor

```
package com.e2eq.framework.util.csv.processors;

import com.e2eq.framework.model.persistent.imports.LookupConfig;
import com.e2eq.framework.model.persistent.imports.LookupFailBehavior;
import com.e2eq.framework.util.csv.LookupService;
import org.supercsv.cellprocessor.CellProcessorAdaptor;
import org.supercsv.cellprocessor.ift.CellProcessor;
import org.supercsv.exception.SuperCsvCellProcessorException;
import org.supercsv.util.CsvContext;

import java.util.Map;
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;

/**
 * Cell processor that looks up values from another collection.
 */
public class LookupProcessor extends CellProcessorAdaptor {

    private final LookupService lookupService;
    private final LookupConfig config;
    private final Map<String, Optional<String>> cache;

    public LookupProcessor(
        LookupService lookupService,
        LookupConfig config,
        CellProcessor next) {
        super(next);
        this.lookupService = lookupService;
        this.config = config;
        this.cache = config.isCacheLookups() ? new ConcurrentHashMap<>() : null;
    }

    @Override
    public Object execute(Object value, CsvContext ctx) {
        if (value == null) {
            return next.execute(null, ctx);
        }

        String key = value.toString();
        Optional<String> result;

        if (cache != null) {
            result = cache.computeIfAbsent(key, k -> lookupService.lookup(config, k));
        } else {
            result = lookupService.lookup(config, key);
        }
    }
}
```

```

        if (result.isEmpty()) {
            switch (config.getOnNotFound()) {
                case FAIL:
                    throw new SuperCsvCellProcessorException(
                        String.format("Lookup failed: no %s found with %s = '%s'",
                            config.getLookupCollection(),
                            config.getLookupMatchField(),
                            key),
                        ctx, this);
                case NULL:
                    return next.execute(null, ctx);
                case PASSTHROUGH:
                    return next.execute(value, ctx);
            }
        }

        return next.execute(result.get(), ctx);
    }

    /**
     * Clear the lookup cache. Call between imports if reusing processor.
     */
    public void clearCache() {
        if (cache != null) {
            cache.clear();
        }
    }
}

```

## 3.4. Enhanced CSVImportHelper

### 3.4.1. New Method Signatures

```

@ApplicationScoped
public class CSVImportHelper {

    @Inject
    Instance<PreValidationTransformer<?>> transformers;

    @Inject
    ImportProfileRepo importProfileRepo;

    @Inject
    LookupService lookupService;

    // ... existing methods unchanged ...

    /**
     * Import CSV with an ImportProfile for transformations.

```

```

*
* @param repo The repository for the target entity type
* @param inputStream The CSV input stream
* @param profile The import profile with transformation rules
* @param formatOptions CSV format options (separator, quote, charset, etc.)
* @param failedRecordHandler Optional handler for failed records
* @return Import result with counts and error details
*/
public <T extends UnversionedBaseModel> ImportResult<T> importCSV(
    BaseMorphiaRepo<T> repo,
    InputStream inputStream,
    ImportProfile profile,
    CSVFormatOptions formatOptions,
    FailedRecordHandler<T> failedRecordHandler);

/**
 * Import CSV with profile referenced by refName.
 *
 * @param repo The repository for the target entity type
 * @param inputStream The CSV input stream
 * @param profileRefName The refName of an existing ImportProfile
 * @param formatOptions CSV format options
 * @param failedRecordHandler Optional handler for failed records
 * @return Import result with counts and error details
 * @throws ValidationException if profile not found
 */
public <T extends UnversionedBaseModel> ImportResult<T> importCSV(
    BaseMorphiaRepo<T> repo,
    InputStream inputStream,
    String profileRefName,
    CSVFormatOptions formatOptions,
    FailedRecordHandler<T> failedRecordHandler);

/**
 * Analyze CSV with profile for preview (session-based flow).
 *
 * @param repo The repository for the target entity type
 * @param inputStream The CSV input stream
 * @param profile The import profile with transformation rules
 * @param formatOptions CSV format options
 * @return Import result with preview data and session ID
 */
public <T extends UnversionedBaseModel> ImportResult<T> analyzeCSV(
    BaseMorphiaRepo<T> repo,
    InputStream inputStream,
    ImportProfile profile,
    CSVFormatOptions formatOptions) throws IOException;
}

```

### 3.4.2. CSVFormatOptions (Refactored)

```
package com.e2eq.framework.util.csv;

import lombok.Builder;
import lombok.Data;

import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.util.List;

/**
 * CSV format configuration options.
 */
@Data
@Builder
public class CSVFormatOptions {

    @Builder.Default
    private char fieldSeparator = ',';

    @Builder.Default
    private char quoteChar = '"';

    @Builder.Default
    private boolean skipHeaderRow = true;

    @Builder.Default
    private Charset charset = StandardCharsets.UTF_8;

    @Builder.Default
    private boolean hasBOM = false;

    @Builder.Default
    private String quotingStrategy = "QUOTE_WHERE_ESSENTIAL";

    /**
     * Column names/fields to import, in CSV column order.
     * If null and profile has columnMappings, uses those.
     * If both null, uses header row.
     */
    private List<String> requestedColumns;
}
```

### 3.4.3. Enhanced Processor Chain Building

```
/**
 * Build cell processors from ImportProfile column mappings.
 */
```

```

private CellProcessor[] buildProcessorsFromProfile(
    Class<?> clazz,
    ImportProfile profile,
    ListCellProcessor listProcessor) {

    List<ColumnMapping> mappings = profile.getColumnMappings();
    CellProcessor[] processors = new CellProcessor[mappings.size()];

    for (int i = 0; i < mappings.size(); i++) {
        ColumnMapping mapping = mappings.get(i);
        processors[i] = buildProcessorChain(clazz, mapping, listProcessor);
    }

    return processors;
}

/**
 * Build a processor chain for a single column.
 * Processors are chained innermost-first (type conversion is innermost).
 */
private CellProcessor buildProcessorChain(
    Class<?> clazz,
    ColumnMapping mapping,
    ListCellProcessor listProcessor) {

    String fieldName = mapping.getTargetField();
    Class<?> type = getFieldType(clazz, fieldName);

    // Start with type-specific processor (innermost)
    CellProcessor chain = buildTypeProcessor(type, mapping, listProcessor);

    // Add lookup if configured
    if (mapping.getLookup() != null) {
        chain = new LookupProcessor(lookupService, mapping.getLookup(), chain);
    }

    // Add value mapping if configured
    if (mapping.getValueMappings() != null && !mapping.getValueMappings().isEmpty()) {
        chain = new ValueMapProcessor(
            mapping.getValueMappings(),
            mapping.isValueMappingCaseSensitive(),
            mapping.getUnmappedValueBehavior(),
            chain);
    }

    // Add regex replacement if configured
    if (mapping.getRegexPattern() != null) {
        chain = new RegexReplaceProcessor(
            mapping.getRegexPattern(),
            mapping.getRegexReplacement(),
            chain);
    }
}

```

```

    }

    // Add case transformation
    if (mapping.getCaseTransform() != CaseTransform.NONE) {
        chain = new CaseTransformProcessor(mapping.getCaseTransform(), chain);
    }

    // Add trim (applied early, before other transformations)
    if (mapping.isTrim()) {
        chain = new org.supercsv.cellprocessor.Trim(chain);
    }

    // Add empty-to-null conversion
    if (mapping.isEmptyToNull()) {
        chain = new EmptyToNullProcessor(chain);
    }

    // Add default value handling (applied last, after null conversion)
    if (mapping.getDefaultValue() != null) {
        chain = new org.supercsv.cellprocessor.ConvertNullTo(mapping.getDefaultValue(
), chain);
    }

    // Wrap with Optional for null safety
    return new org.supercsv.cellprocessor.Optional(chain);
}

private CellProcessor buildTypeProcessor(
    Class<?> type,
    ColumnMapping mapping,
    ListCellProcessor listProcessor) {

    if (mapping.getTargetField().contains("[") {
        return listProcessor;
    }

    // Handle date with custom format
    if (type == java.time.LocalDate.class || type == java.time.LocalDateTime.class
        || type == java.util.Date.class) {
        if (mapping.getDateFormat() != null) {
            return new ParseDateProcessor(mapping.getDateFormat(), mapping.getLocale(
));
        }
    }

    // Standard type processors
    if (type == int.class || type == Integer.class) {
        return new org.supercsv.cellprocessor.ParseInt();
    } else if (type == long.class || type == Long.class) {
        return new org.supercsv.cellprocessor.ParseLong();
    } else if (type == double.class || type == Double.class)

```

```

        || type == float.class || type == Float.class) {
            return new org.supercsv.cellprocessor.ParseDouble();
        } else if (type == java.math.BigDecimal.class) {
            return new org.supercsv.cellprocessor.ParseBigDecimal();
        } else if (type == boolean.class || type == Boolean.class) {
            return new org.supercsv.cellprocessor.ParseBool();
        } else if (type != null && type.isEnum()) {
            return new ParseEnum((Class<? extends Enum<?>>) type);
        }

        // Default: pass through as string
        return new org.supercsv.cellprocessor.constraint.NotNull();
    }
}

```

### 3.4.4. PreValidationTransformer Integration

```

/**
 * Apply pre-validation transformers to a parsed bean.
 */
@SuppressWarnings("unchecked")
private <T extends UnversionedBaseModel> T applyTransformers(
    T bean,
    ImportRowContext context) throws TransformationException {

    if (transformers == null || transformers.isUnsatisfied()) {
        return bean;
    }

    // Collect and sort transformers by priority
    List<PreValidationTransformer<T>> applicable = new ArrayList<>();
    for (PreValidationTransformer<?> transformer : transformers) {
        if (transformer.getTargetType().isAssignableFrom(bean.getClass())) {
            applicable.add((PreValidationTransformer<T>) transformer);
        }
    }

    applicable.sort(Comparator.comparingInt(PreValidationTransformer::getPriority));

    // Apply transformers in order
    T result = bean;
    for (PreValidationTransformer<T> transformer : applicable) {
        result = transformer.transform(result, context);
    }

    return result;
}

```



### 3.4.5. Intent Resolution

When an `intentColumn` is configured, the import process reads the intent from each row and validates it.

```
/**
 * Resolve the intent for a row from CSV data or profile defaults.
 *
 * @param profile The import profile (may have intentColumn configured)
 * @param rowValues The parsed CSV row values by column name
 * @param existsInDb Whether the record already exists (for UPSERT resolution)
 * @return The resolved intent for this row
 * @throws ImportException if an invalid intent value is specified
 */
private ImportIntent resolveIntent(
    ImportProfile profile,
    Map<String, String> rowValues,
    boolean existsInDb) {

    // If no intent column configured, use default behavior
    if (profile == null || profile.getIntentColumn() == null) {
        return existsInDb ? ImportIntent.UPDATE : ImportIntent.INSERT;
    }

    // Read intent from CSV column
    String intentValue = rowValues.get(profile.getIntentColumn());

    if (intentValue != null && !intentValue.isBlank()) {
        String normalized = intentValue.toUpperCase().trim();

        // Validate against allowed intents (explicitly reject MERGE/DELETE)
        switch (normalized) {
            case "INSERT":
                return ImportIntent.INSERT;
            case "UPDATE":
                return ImportIntent.UPDATE;
            case "SKIP":
                return ImportIntent.SKIP;
            case "UPSERT":
                return existsInDb ? ImportIntent.UPDATE : ImportIntent.INSERT;
            case "MERGE":
            case "DELETE":
                throw new ImportException(
                    "Intent '" + normalized + "' is not supported. " +
                    "Only INSERT, UPDATE, UPSERT, and SKIP are allowed.");
            default:
                throw new ImportException(
                    "Invalid intent value: '" + intentValue + "'. " +
                    "Valid values are: INSERT, UPDATE, UPSERT, SKIP.");
        }
    }
}
```

```

}

// Fall back to profile default
String defaultIntent = profile.getDefaultIntent();
if (defaultIntent == null || defaultIntent.isBlank()) {
    defaultIntent = "UPSERT";
}

return switch (defaultIntent.toUpperCase()) {
    case "INSERT" -> ImportIntent.INSERT;
    case "UPDATE" -> ImportIntent.UPDATE;
    case "SKIP" -> ImportIntent.SKIP;
    default -> existsInDb ? ImportIntent.UPDATE : ImportIntent.INSERT; // UPSERT
};
}

```

### 3.4.6. Intent Execution

```

/**
 * Execute the save operation based on the resolved intent.
 */
private <T extends UnversionedBaseModel> void executeWithIntent(
    BaseMorphiaRepo<T> repo,
    T bean,
    ImportIntent intent,
    ImportRowResult<T> rowResult) {

    switch (intent) {
        case INSERT -> {
            // Verify record doesn't exist
            Optional<T> existing = repo.findByRefName(bean.getRefName());
            if (existing.isPresent()) {
                throw new ImportException(
                    "Cannot INSERT: record with refName '" + bean.getRefName() +
                    "' already exists. Use UPDATE or UPSERT intent.");
            }
            repo.save(bean);
            rowResult.setIntent(Intent.INSERT);
        }

        case UPDATE -> {
            // Verify record exists and get its ID
            Optional<T> existing = repo.findByRefName(bean.getRefName());
            if (existing.isEmpty()) {
                throw new ImportException(
                    "Cannot UPDATE: record with refName '" + bean.getRefName() +
                    "' does not exist. Use INSERT or UPSERT intent.");
            }
            // Copy ID to ensure update, not insert

```

```

        bean.setId(existing.get().getId());
        repo.save(bean);
        rowResult.setIntent(Intent.UPDATE);
    }

    case SKIP -> {
        // Do nothing - row is intentionally skipped
        rowResult.setIntent(Intent.SKIP);
    }

    // UPSERT is resolved to INSERT or UPDATE before this method is called
}
}

```

## 3.5. REST API Enhancements

### 3.5.1. Updated Endpoints

```

// In BaseResource.java

@POST
@Path("/csv")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_JSON)
@Operation(summary = "Import a list of entities from a CSV file")
public Response importCSVList(
    @Context UriInfo info,
    @BeanParam FileUpload fileUpload,
    // Existing parameters
    @QueryParam("fieldSeparator") @DefaultValue(",") String fieldSeparator,
    @QueryParam("quoteChar") @DefaultValue "\"" String quoteChar,
    @QueryParam("skipHeaderRow") @DefaultValue("true") boolean skipHeaderRow,
    @QueryParam("requestedColumns") String requestedColumns,
    @QueryParam("charsetEncoding") @DefaultValue("UTF-8-without-BOM") String
charsetEncoding,
    @QueryParam("quotingStrategy") @DefaultValue("QUOTE_WHERE_ESSENTIAL") String
quotingStrategy,
    // NEW parameters
    @Parameter(description = "RefName of ImportProfile to use for value mapping
and transformations")
    @QueryParam("profileRefName") String profileRefName,
    @Parameter(description = "CSV column name containing per-row intent (INSERT,
UPDATE, SKIP). " +
                                "If not specified, uses UPSERT behavior (auto-
detect).")
    @QueryParam("intentColumn") String intentColumn,
    @Parameter(description = "Default intent when intentColumn value is empty. " +
                                "One of: INSERT, UPDATE, UPSERT, SKIP. Default:
UPSERT")

```

```

        @QueryParam("defaultIntent") @DefaultValue("UPSERT") String defaultIntent,
        @Parameter(description = "Enable header modifier parsing. When true, headers
like " +
                                "'name*' (required), 'desc?' (optional), 'createdAt~'
(calculated), " +
                                "'sku#' (key field) are parsed for field metadata.")
        @QueryParam("enableHeaderModifiers") @DefaultValue("false") boolean
enableHeaderModifiers
    ) {
        // Implementation uses profile if provided, or parameters directly
    }

@POST
@Path("csv/session")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_JSON)
@Operation(summary = "Create an import session with preview")
public Response createCsvImportSession(
    // ... same parameters including profileRefName ...
) {
    // Implementation
}

```

### 3.5.2. ImportProfile Resource

```

package com.e2eq.framework.rest.resources;

import com.e2eq.framework.model.persistent.imports.ImportProfile;
import com.e2eq.framework.model.persistent.morphia.ImportProfileRepo;
import jakarta.ws.rs.Path;

/**
 * REST resource for managing ImportProfiles.
 */
@Path("/integration/import-profiles")
public class ImportProfileResource extends BaseResource<ImportProfile,
ImportProfileRepo> {
    // Standard CRUD endpoints inherited from BaseResource
}

```

## 3.6. Example Usage

### 3.6.1. Example ImportProfile (JSON)

```

{
  "refName": "product-import-v1",
  "displayName": "Product Import Profile",

```

```

"description": "Maps legacy product CSV format to Product entity",
"targetType": "com.example.Product",
"columnMappings": [
  {
    "sourceColumn": "SKU",
    "targetField": "refName",
    "trim": true,
    "caseTransform": "UPPER"
  },
  {
    "sourceColumn": "Product Name",
    "targetField": "displayName",
    "trim": true
  },
  {
    "sourceColumn": "Status",
    "targetField": "status",
    "valueMappings": {
      "A": "ACTIVE",
      "I": "INACTIVE",
      "D": "DISCONTINUED",
      "P": "PENDING"
    },
    "unmappedValueBehavior": "FAIL"
  },
  {
    "sourceColumn": "Active",
    "targetField": "isActive",
    "valueMappings": {
      "Y": "true",
      "N": "false",
      "YES": "true",
      "NO": "false",
      "1": "true",
      "0": "false"
    },
    "valueMappingCaseSensitive": false
  },
  {
    "sourceColumn": "Category",
    "targetField": "categoryRefName",
    "lookup": {
      "lookupCollection": "Category",
      "lookupMatchField": "displayName",
      "lookupReturnField": "refName",
      "onNotFound": "FAIL",
      "cacheLookups": true
    }
  },
  {
    "sourceColumn": "Price",

```

```

    "targetField": "price",
    "regexPattern": "[$,]",
    "regexReplacement": "",
    "trim": true
  },
  {
    "sourceColumn": "Tags",
    "targetField": "tags",
    "regexPattern": "\\s*[;,]\\s*",
    "regexReplacement": ","
  }
],
"globalTransformations": {
  "trimStrings": true,
  "emptyStringsToNull": true,
  "normalizeWhitespace": true
},
"intentColumn": "_action",
"defaultIntent": "UPSERT"
}

```

### 3.6.2. Example CSV (without intent column)

```

SKU,Product Name,Status,Active,Category,Price,Tags
sku-001 ,Widget Pro,A,Y,Electronics,$19.99,new; featured
SKU-002,Gadget Plus,I,N,Home & Garden,$29.99,sale, clearance
sku-003,Tool Kit,D,0,Tools,$49.99,

```

### 3.6.3. Example CSV (with intent column)

When you need explicit control over whether each row is inserted, updated, or skipped:

```

_action,SKU,Product Name,Status,Active,Category,Price,Tags
INSERT,SKU-NEW-001,Brand New Widget,A,Y,Electronics,$19.99,new
UPDATE,SKU-002,Updated Gadget Name,A,Y,Home & Garden,$34.99,updated
SKIP,SKU-003,Ignore This Row,I,N,Tools,$0.00,
UPSERT,SKU-004,Auto Detect Mode,A,Y,Electronics,$24.99,auto
INSERT,SKU-NEW-002,Another New Product,A,Y,Tools,$49.99,new

```

#### Intent column behavior:

| Intent | Behavior   |
|--------|--|
| INSERT | Insert new record. <b>Fails if refName already exists.</b>       |
| UPDATE | Update existing record. <b>Fails if refName does not exist.</b>  |
| UPSERT | Auto-detect: INSERT if new, UPDATE if exists. (Default behavior) |

| Intent | Behavior  |
|--------|---|
| SKIP   | Skip this row entirely - no database operation. |
| MERGE  | <b>Not supported</b> - returns error.           |
| DELETE | <b>Not supported</b> - returns error.           |

### 3.6.4. API Calls

```
# Create the profile first
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  "https://host/api/integration/import-profiles" \
  -d @product-import-profile.json

# Import using the profile (UPSERT behavior - auto-detect insert vs update)
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -F "file=@products.csv" \
  "https://host/api/products/csv?profileRefName=product-import-v1&skipHeaderRow=true"

# Import with explicit intent column (no profile needed for intent)
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -F "file=@products-with-intent.csv" \

"https://host/api/products/csv?requestedColumns=_action,refName,displayName,status&intentColumn=_action&skipHeaderRow=true"

# Import with intent column and profile for value mappings
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -F "file=@products-with-intent.csv" \
  "https://host/api/products/csv?profileRefName=product-import-v1&intentColumn=_action&skipHeaderRow=true"

# Import with default intent (all rows treated as INSERT, fail if exists)
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -F "file=@new-products.csv" \
  "https://host/api/products/csv?profileRefName=product-import-v1&defaultIntent=INSERT&skipHeaderRow=true"
```

### 3.6.5. Example PreValidationTransformer

```
package com.example.imports;

import com.e2eq.framework.util.csv.ImportRowContext;
```

```

import com.e2eq.framework.util.csv.PreValidationTransformer;
import com.e2eq.framework.util.csv.TransformationException;
import com.example.model.Product;
import jakarta.enterprise.context.ApplicationScoped;

import java.math.BigDecimal;

@ApplicationScoped
public class ProductTransformer implements PreValidationTransformer<Product> {

    @Override
    public Class<Product> getTargetType() {
        return Product.class;
    }

    @Override
    public int getPriority() {
        return 500; // Run before default transformers
    }

    @Override
    public Product transform(Product product, ImportRowContext context)
        throws TransformationException {

        // Compute derived fields
        if (product.getPrice() != null && product.getQuantity() != null) {
            product.setTotalValue(
                product.getPrice().multiply(BigDecimal.valueOf(product.getQuantity()))
            );
        }

        // Business rule: new products default to DRAFT status
        if (product.getStatus() == null) {
            product.setStatus(ProductStatus.DRAFT);
        }

        // Cross-field validation
        if (product.getDiscountPrice() != null && product.getPrice() != null) {
            if (product.getDiscountPrice().compareTo(product.getPrice()) > 0) {
                throw new TransformationException(
                    "discountPrice",
                    "Discount price cannot exceed regular price"
                );
            }
        }

        // Conditional transformation
        if ("CLEARANCE".equals(product.getCategory())) {
            product.setFeatured(false);
            product.setSearchable(false);
        }
    }
}

```



```
    return product;  
  }  
}
```

# Chapter 4. Migration Path

## 4.1. Phase 1: Core Models (Non-Breaking)

1. Add `ImportProfile` entity and repository
2. Add `ColumnMapping`, `LookupConfig`, `GlobalTransformations`, `CaseTransform`, enums
3. Add `LookupService` interface
4. Create `ImportProfileResource` REST endpoint

**Backward Compatibility:** No changes to existing behavior.

## 4.2. Phase 2: Cell Processors (Non-Breaking)

1. Add `ValueMapProcessor`
2. Add `RegexReplaceProcessor`
3. Add `CaseTransformProcessor`
4. Add `LookupProcessor`
5. Add `EmptyToNullProcessor`

**Backward Compatibility:** New processors only used when profile specified.

## 4.3. Phase 3: CSVImportHelper Enhancement (Non-Breaking)

1. Add overloaded methods accepting `ImportProfile`
2. Add `buildProcessorsFromProfile()` method
3. Implement processor chain building from profile
4. Add `LookupService` implementation

**Backward Compatibility:** Existing methods unchanged; new methods are additions.

## 4.4. Phase 4: PreValidationTransformer SPI (Non-Breaking)

1. Add `PreValidationTransformer` interface
2. Add `ImportRowContext` class
3. Add `TransformationException`
4. Integrate transformer invocation into import flow

**Backward Compatibility:** Transformers are optional; empty by default.

## 4.5. Phase 5: Header Modifiers (Non-Breaking)

1. Add `HeaderModifier` enum and `ParsedHeader` class
2. Add `FieldCalculator` SPI interface
3. Add built-in calculators (Timestamp, UUID, RefName)
4. Add `InlineFieldCalculator` for profile-defined calculations
5. Add `enableHeaderModifiers` flag to `ImportProfile`
6. Add `enableHeaderModifiers` REST parameter
7. Integrate header parsing into import flow

**Backward Compatibility:** Feature is opt-in; disabled by default.

## 4.6. Phase 6: REST API Updates (Non-Breaking)

1. Add `profileRefName` query parameter to CSV endpoints
2. Add `intentColumn` and `defaultIntent` parameters
3. Add `enableHeaderModifiers` parameter
4. Update OpenAPI documentation
5. Add user guide documentation

**Backward Compatibility:** All parameters are optional; existing calls work unchanged.

# Chapter 5. Security Considerations

## 5.1. Profile Access Control

- `ImportProfile` entities follow standard `DataDomain` scoping
- Profiles can only be used within their tenant/org context
- Cross-tenant profile access is blocked

## 5.2. Lookup Security

- Lookup queries respect the caller's security context
- `RuleContext` filters are applied to lookup queries
- Cross-tenant lookups are blocked by `DataDomain` filtering
- Lookups only access collections the user has `VIEW` permission on

## 5.3. Regex Safety

- Consider limiting regex complexity to prevent ReDoS attacks
- Implement timeout for regex operations on large values
- Log warnings for potentially dangerous patterns

# Chapter 6. Performance Considerations

## 6.1. Lookup Caching

- Lookups are cached per-import session by default (`cacheLookups: true`)
- Cache is cleared between import sessions
- Consider LRU eviction for imports with many unique lookup values

## 6.2. Batch Lookup Optimization

- `LookupService.batchLookup()` allows batch resolution
- Collect all values for a column, resolve in single query
- Reduces database round-trips for large imports

## 6.3. Profile Caching

- Frequently used profiles could be cached in memory
- Consider cache invalidation on profile update

# Chapter 7. Testing Strategy

## 7.1. Unit Tests

- Test each cell processor in isolation
- Test processor chain building with various configurations
- Test value mapping with case sensitivity options
- Test lookup with various failure behaviors
- Test PreValidationTransformer priority ordering
- Test header modifier parsing (\*, ?, ~, #)
- Test ParsedHeader.parse() with various inputs
- Test FieldCalculator invocation for calculated fields
- Test required field validation
- Test key field intent detection

## 7.2. Integration Tests

- Test full import flow with profiles
- Test lookup resolution across collections
- Test security filtering on lookups
- Test error handling and reporting
- Test header modifiers with real CSV files
- Test backward compatibility (modifiers disabled)
- Test calculated fields with CDI-discovered calculators
- Test inline calculators from profile

## 7.3. Performance Tests

- Benchmark imports with 100K+ rows
- Measure lookup cache effectiveness
- Profile memory usage during large imports

# Chapter 8. Appendix A: Common Value Mapping Patterns

## 8.1. Boolean Values

```
{
  "valueMappings": {
    "Y": "true", "N": "false",
    "YES": "true", "NO": "false",
    "TRUE": "true", "FALSE": "false",
    "1": "true", "0": "false",
    "T": "true", "F": "false",
    "ON": "true", "OFF": "false"
  },
  "valueMappingCaseSensitive": false
}
```

## 8.2. Status Codes

```
{
  "valueMappings": {
    "10": "DRAFT",
    "20": "PENDING",
    "30": "APPROVED",
    "40": "ACTIVE",
    "50": "SUSPENDED",
    "90": "CLOSED",
    "99": "CANCELLED"
  }
}
```

## 8.3. Country Codes

```
{
  "valueMappings": {
    "USA": "US",
    "United States": "US",
    "UK": "GB",
    "United Kingdom": "GB",
    "Great Britain": "GB"
  },
  "valueMappingCaseSensitive": false,
  "unmappedValueBehavior": "PASSTHROUGH"
}
```

}



# Chapter 9. Appendix B: Regex Patterns

## 9.1. Remove Currency Symbols

```
{  
  "regexPattern": "[${€£¥,}]",  
  "regexReplacement": ""  
}
```

## 9.2. Normalize Phone Numbers

```
{  
  "regexPattern": "[^0-9+]",  
  "regexReplacement": ""  
}
```

## 9.3. Extract Numeric ID

```
{  
  "regexPattern": "^([A-Z]+)-([0-9]+)$",  
  "regexReplacement": "$1"  
}
```

## 9.4. Normalize Whitespace

```
{  
  "regexPattern": "\\s+",  
  "regexReplacement": " "  
}
```

# Chapter 10. Implementation Status

The following components have been implemented:

## 10.1. Completed Components

| Component                | Package                                     | Description                                       |
|--------------------------|---|---|
| CaseTransform            | com.e2eq.framework.model.persistent.imports | Enum for string case transformation options       |
| UnmappedValueBehavior    | com.e2eq.framework.model.persistent.imports | Enum for behavior when CSV value not in mappings  |
| LookupFailBehavior       | com.e2eq.framework.model.persistent.imports | Enum for behavior when lookup fails               |
| LookupConfig             | com.e2eq.framework.model.persistent.imports | Configuration for cross-collection lookups        |
| ColumnMapping            | com.e2eq.framework.model.persistent.imports | Per-column transformation configuration           |
| GlobalTransformations    | com.e2eq.framework.model.persistent.imports | Global string transformation settings             |
| InlineFieldCalculator    | com.e2eq.framework.model.persistent.imports | Inline calculator definitions for profiles        |
| HeaderModifier           | com.e2eq.framework.model.persistent.imports | Enum for header modifier suffixes (*, ?, ~, #)    |
| ParsedHeader             | com.e2eq.framework.model.persistent.imports | Parsed CSV header with modifier                   |
| ImportIntent             | com.e2eq.framework.model.persistent.imports | Import intent enum (INSERT, UPDATE, SKIP, UPSERT) |
| ImportProfile            | com.e2eq.framework.model.persistent.imports | Persistent profile entity extending BaseModel     |
| ImportSession (updated)  | com.e2eq.framework.model.persistent.imports | Added <code>profileRefName</code> field           |
| ImportProfileRepo        | com.e2eq.framework.model.persistent.morphia | Repository for ImportProfile entities             |
| FieldCalculator          | com.e2eq.framework.imports.spi              | SPI interface for field calculators               |
| PreValidationTransformer | com.e2eq.framework.imports.spi              | SPI interface for pre-validation transformers     |
| ImportContext            | com.e2eq.framework.imports.spi              | Context object for SPI implementations            |
| LookupService            | com.e2eq.framework.imports.service          | Interface for cross-collection lookups            |
| LookupServiceImpl        | com.e2eq.framework.imports.service          | Default implementation with caching               |

| Component                  | Package                                | Description  |
|----------------------------|--|--|
| ImportProfileService       | com.e2eq.framework.imports.service     | Orchestration service for profile processing       |
| TimestampFieldCalculator   | com.e2eq.framework.imports.calculators | Built-in calculator for timestamp fields           |
| UUIDFieldCalculator        | com.e2eq.framework.imports.calculators | Built-in calculator for UUID fields                |
| RefNameFieldCalculator     | com.e2eq.framework.imports.calculators | Built-in calculator for refName generation         |
| ValueMapProcessor          | com.e2eq.framework.imports.processors  | CellProcessor for value mapping                    |
| RegexReplaceProcessor      | com.e2eq.framework.imports.processors  | CellProcessor for regex replacement                |
| CaseTransformProcessor     | com.e2eq.framework.imports.processors  | CellProcessor for case transformation              |
| LookupProcessor            | com.e2eq.framework.imports.processors  | CellProcessor for cross-collection lookups         |
| GlobalTransformProcessor   | com.e2eq.framework.imports.processors  | CellProcessor for global transformations           |
| RowValueResolver           | com.e2eq.framework.imports.spi         | SPI interface for arbitrary per-row code execution |
| RowValueResolverProcessor  | com.e2eq.framework.imports.processors  | CellProcessor wrapper for RowValueResolver         |
| CSVImportHelper (enhanced) | com.e2eq.framework.util                | Added <code>analyzeCSVWithProfile()</code> method  |
| BaseResource (enhanced)    | com.e2eq.framework.rest.resources      | Added <code>/csv/session/profile</code> endpoint   |

## 10.2. REST API Endpoints

| Method | Path   | Description  |
|--------|--|--|
| POST   | <code>/{"entity"}/csv/session/profile</code> | Create import session with ImportProfile support. Accepts <code>profileRefName</code> query parameter. |

## 10.3. Usage

To use the new profile-based import:

1. **Create an ImportProfile** via the standard BaseResource CRUD endpoints or directly in the database
2. **Call the profile endpoint:**

```
curl -X POST \  
  -H "Authorization: Bearer $TOKEN" \  
  -F "file=@data.csv" \  
  "https://host/api/products/csv/session/profile?profileRefName=my-profile"
```

3. **Review the preview** and commit using the existing session workflow:

```
# Commit the session  
curl -X POST \  
  -H "Authorization: Bearer $TOKEN" \  
  "https://host/api/products/csv/session/{sessionId}/commit"
```

# Chapter 11. RowValueResolver: Per-Row Arbitrary Code Execution

The `RowValueResolver` SPI allows you to run arbitrary code for each row during CSV import. Unlike static lookups which match a single field against a collection, RowValueResolvers have access to **all columns in the row** and can invoke external services, apply complex business logic, or perform conditional transformations.

## 11.1. Use Cases

- **Geocoding addresses:** Look up latitude/longitude from a geocoding service based on address fields
- **Cross-field validation:** Validate combinations of fields that depend on each other
- **External service calls:** Call APIs to enrich, validate, or transform data
- **Complex business rules:** Apply conditional logic based on multiple fields
- **Composite key generation:** Generate identifiers from multiple source columns

## 11.2. Example: Address Geocoding

This example shows how to geocode addresses during import, setting latitude and longitude fields based on an address lookup service.

### 11.2.1. 1. Create the RowValueResolver Implementation

```
package com.example.imports;

import com.e2eq.framework.imports.spi.ImportContext;
import com.e2eq.framework.imports.spi.RowValueResolver;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.inject.Named;

import java.util.Map;

@ApplicationScoped
@Named("geocodeResolver")
public class GeocodeResolver implements RowValueResolver {

    @Inject
    GeocodingService geocodingService;

    @Override
    public String getName() {
        return "geocodeResolver";
    }
}
```

```

@Override
public ResolveResult resolve(String inputValue, Map<String, Object> rowData,
ImportContext context) {
    // Access multiple address columns from the row
    String street = getString(rowData, "street");
    String city = getString(rowData, "city");
    String state = getString(rowData, "state");
    String zip = getString(rowData, "zip");
    String country = getString(rowData, "country");

    // Build full address for geocoding
    String fullAddress = String.format("%s, %s, %s %s, %s",
        street, city, state, zip, country);

    try {
        // Call external geocoding service
        GeocodingResult result = geocodingService.geocode(fullAddress);

        if (result == null || !result.isValid()) {
            // Return null - field won't be set but row continues
            return ResolveResult.nullValue();
        }

        // Return a GeoPoint or coordinate object
        return ResolveResult.success(new GeoPoint(result.getLatitude(), result
.getLongitude()));

    } catch (GeocodingException e) {
        // Log warning but don't fail the row
        return ResolveResult.nullValue();
    } catch (Exception e) {
        // Critical error - skip this row
        return ResolveResult.error("Geocoding failed: " + e.getMessage());
    }
}

private String getString(Map<String, Object> rowData, String key) {
    Object value = rowData.get(key);
    return value != null ? value.toString().trim() : "";
}
}

```

### 11.2.2. 2. Configure the ImportProfile

```

{
  "refName": "location-import-profile",
  "displayName": "Location Import with Geocoding",
  "targetCollection": "com.example.model.Location",

```

```

"columnMappings": [
  {
    "sourceColumn": "name",
    "targetField": "displayName"
  },
  {
    "sourceColumn": "street",
    "targetField": "address.street"
  },
  {
    "sourceColumn": "city",
    "targetField": "address.city"
  },
  {
    "sourceColumn": "state",
    "targetField": "address.state"
  },
  {
    "sourceColumn": "zip",
    "targetField": "address.postalCode"
  },
  {
    "sourceColumn": "country",
    "targetField": "address.country",
    "defaultValue": "USA"
  },
  {
    "sourceColumn": "street",
    "targetField": "location",
    "rowValueResolverName": "geocodeResolver"
  }
]
}

```

Note: The `sourceColumn` for the `geocodeResolver` can be any column - the resolver has access to all row data via the `rowData` map. The column value is passed as `inputValue` but you can ignore it and use any combination of columns.

### 11.2.3. 3. Sample CSV Input

```

name,street,city,state,zip,country
"Acme Corp HQ","123 Main St","San Francisco","CA","94102","USA"
"East Coast Office","456 Park Ave","New York","NY","10022","USA"
"European Branch","10 Oxford St","London","","W1D 1BS","UK"

```

### 11.2.4. 4. Result

After import, each Location entity will have:

- All address fields populated from the CSV
- A **location** field containing a GeoPoint with latitude/longitude

## 11.3. RowValueResolver API

```
public interface RowValueResolver {

    /**
     * Unique name for this resolver (used in ColumnMapping.rowValueResolverName).
     */
    String getName();

    /**
     * Resolve a value based on input and row data.
     *
     * @param inputValue the value from the CSV column (after prior transformations)
     * @param rowData all column values from the current row (column name -> value)
     * @param context import context with profile, realm, row number, session ID
     * @return the resolution result
     */
    ResolveResult resolve(String inputValue, Map<String, Object> rowData,
        ImportContext context);

    /**
     * Check if this resolver applies to the given target class.
     * Default returns true (applies to all types).
     */
    default boolean appliesTo(Class<?> targetClass) {
        return true;
    }

    /**
     * Priority order - lower values run first. Default is 100.
     */
    default int getOrder() {
        return 100;
    }
}
```

## 11.4. ResolveResult Options

| Method  | Description  |
|---|--|
| <code>ResolveResult.success(value)</code>             | Return a resolved value to set on the target field |
| <code>ResolveResult.passthrough(originalValue)</code> | Keep the original value unchanged                  |



| Method                                    | Description                                     |
|---|---|
| <code>ResolveResult.nullValue()</code>    | Set the field to null (but continue processing) |
| <code>ResolveResult.skip()</code>         | Skip this row entirely (not an error)           |
| <code>ResolveResult.skip(reason)</code>   | Skip with a reason message                      |
| <code>ResolveResult.error(message)</code> | Mark row as error with the given message        |

## 11.5. Processing Order

RowValueResolvers are applied after field calculators but before pre-validation transformers:

1. CSV parsing with CellProcessors (value mapping, regex, case transform, static lookups)
2. Field calculators (timestamps, UUIDs, templates)
3. **RowValueResolvers** (arbitrary code with full row access)
4. Pre-validation transformers
5. Bean validation