Phase 3: Intelligent Data Migration Engine - TWIST ERP

Implementation Guide

Duration: 6–8 weeks
Version: 1.0
Date: October 2025
Project: TWIST ERP - Visual Drag-and-Drop Multi-Company ERP

1. Phase Overview

Phase 3 implements the intelligent data migration engine - the most critical success factor for TWIST ERP adoption. This system enables SMEs to migrate from Excel, CSV files, and legacy databases to TWIST ERP with minimal effort through AI-assisted field mapping, validation, and data cleansing.

Key Objectives

- Build AI-powered column/field mapping engine
- Implement data profiling and quality assessment
- Create interactive mapping interface with drag-and-drop
- Develop multi-company batch import support
- Enable incremental and parallel migrations
- Implement rollback and audit capabilities
- Create reusable template library
- Support Excel, CSV, and database sources

Success Criteria

- 90%+ accuracy in automatic field mapping
- Migration time reduced by 70% vs manual entry
- Zero data loss during migration
- Support files up to 100,000 rows
- Template reuse across similar imports
- Full audit trail for all migrations

2. Architecture Overview

2.1 Migration Pipeline

```
┌─────────────────┐
│  Data Source    │ (Excel, CSV, Database)
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│  File Upload    │ Parse &amp; Profile
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│ AI Field Match  │ Smart Mapping Suggestions
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│  User Review    │ Drag-and-Drop Mapping
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│  Validation     │ Rules &amp; Data Quality
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│ Transformation  │ Cleansing &amp; Normalization
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│    Import       │ Transaction with Rollback
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│  Audit Log      │ Complete History
└─────────────────┘
```

2.2 Component Stack

- **File Processing:** pandas, openpyxl, xlrd
- **AI Matching:** scikit-learn, fuzzy-wuzzy, sentence-transformers
- **Validation:** pydantic, cerberus
- **Task Queue:** Celery for async processing
- **Storage:** PostgreSQL + File Storage (S3/Local)

- **Frontend:** React with drag-and-drop UI

3. Data Models

3.1 Migration Session Models

```python
# backend/apps/data_migration/models/session.py
from django.db import models
from shared.models import CompanyAwareModel
import json

class MigrationSession(CompanyAwareModel):
    """
    Tracks complete data migration session
    """
    session_id = models.UUIDField(unique=True, editable=False)
    name = models.CharField(max_length=255)
    description = models.TextField(blank=True)

    # Source info
    source_type = models.CharField(
        max_length=20,
        choices=[
            ('EXCEL', 'Excel File'),
            ('CSV', 'CSV File'),
            ('DATABASE', 'Database'),
            ('API', 'API Import'),
        ]
    )
    source_file = models.FileField(
        upload_to='migrations/%Y/%m/',
        null=True,
        blank=True
    )
    source_connection = models.JSONField(
        null=True,
        blank=True,
        help_text="Database connection info"
    )

    # Target
    target_module = models.CharField(max_length=50)
    target_model = models.CharField(max_length=50)

    # Status
    status = models.CharField(
        max_length=20,
        choices=[
            ('UPLOADED', 'File Uploaded'),
            ('PROFILED', 'Data Profiled'),
            ('MAPPED', 'Fields Mapped'),
            ('VALIDATED', 'Validation Complete'),
            ('IMPORTING', 'Import in Progress'),
            ('COMPLETED', 'Completed'),
            ('FAILED', 'Failed'),
            ('ROLLED_BACK', 'Rolled Back'),
        ],
        default='UPLOADED'
    )

    # Statistics
    total_rows = models.IntegerField(default=0)
    processed_rows = models.IntegerField(default=0)
    success_rows = models.IntegerField(default=0)
    error_rows = models.IntegerField(default=0)
    skipped_rows = models.IntegerField(default=0)

    # Configuration
    mapping_config = models.JSONField(default=dict)
    validation_rules = models.JSONField(default=dict)
    transformation_rules = models.JSONField(default=dict)

    # Template
    template = models.ForeignKey(
        'MigrationTemplate',
        on_delete=models.SET_NULL,
        null=True,
        blank=True
    )
    save_as_template = models.BooleanField(default=False)

    # Audit
    started_at = models.DateTimeField(null=True, blank=True)
    completed_at = models.DateTimeField(null=True, blank=True)

    class Meta:
        ordering = ['-created_at']
        indexes = [
            models.Index(fields=['company', 'status']),
            models.Index(fields=['company', 'target_module']),
        ]

class MigrationTemplate(CompanyAwareModel):
    """
    Reusable migration templates
    """
    name = models.CharField(max_length=255)
    description = models.TextField(blank=True)

    target_module = models.CharField(max_length=50)
    target_model = models.CharField(max_length=50)

    # Template configuration
    field_mappings = models.JSONField(default=dict)
    validation_rules = models.JSONField(default=dict)
    transformation_rules = models.JSONField(default=dict)
    default_values = models.JSONField(default=dict)

    # Usage stats
    usage_count = models.IntegerField(default=0)
    last_used_at = models.DateTimeField(null=True, blank=True)

    is_public = models.BooleanField(
        default=False,
        help_text="Available to all users in company"
    )

    class Meta:
        unique_together = [['company', 'name', 'target_model']]

class DataProfile(models.Model):
    """
    Source data profiling results
    """
    session = models.OneToOneField(
        MigrationSession,
        on_delete=models.CASCADE,
        related_name='profile'
    )

    # Column analysis
    columns = models.JSONField(default=list)
    column_types = models.JSONField(default=dict)
    column_stats = models.JSONField(default=dict)

    # Data quality
    null_counts = models.JSONField(default=dict)
    unique_counts = models.JSONField(default=dict)
    duplicate_rows = models.IntegerField(default=0)

    # Sample data
    sample_rows = models.JSONField(default=list)

    profiled_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        db_table = 'migration_data_profiles'
```

3.2 Migration Results Tracking

```
# backend/apps/data_migration/models/results.py
from django.db import models

class MigrationLog(models.Model):
    """
    Detailed log of migration operations
    """
    session = models.ForeignKey(
        'MigrationSession',
        on_delete=models.CASCADE,
        related_name='logs'
    )

    log_level = models.CharField(
        max_length=10,
        choices=[
            ('INFO', 'Info'),
            ('WARNING', 'Warning'),
            ('ERROR', 'Error'),
        ]
    )

    message = models.TextField()
    row_number = models.IntegerField(null=True, blank=True)
    details = models.JSONField(null=True, blank=True)

    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['created_at']
        indexes = [
            models.Index(fields=['session', 'log_level']),
        ]

class MigrationError(models.Model):
    """
    Track errors during migration
    """
    session = models.ForeignKey(
        'MigrationSession',
        on_delete=models.CASCADE,
        related_name='errors'
    )

    row_number = models.IntegerField()
    source_data = models.JSONField()

    error_type = models.CharField(max_length=50)
    error_message = models.TextField()
    field_name = models.CharField(max_length=100, blank=True)

    can_retry = models.BooleanField(default=True)
    is_resolved = models.BooleanField(default=False)

    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        indexes = [
            models.Index(fields=['session', 'is_resolved']),
        ]

class MigrationRecord(models.Model):
    """
    Tracks imported records for rollback
    """
    session = models.ForeignKey(
        'MigrationSession',
        on_delete=models.CASCADE,
        related_name='records'
    )
    target_model = models.CharField(max_length=50)
    target_id = models.IntegerField()

    source_row_number = models.IntegerField()
    source_data = models.JSONField()

    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        indexes = [
            models.Index(fields=['session', 'target_model']),
        ]
```

4. AI Field Mapping Engine

4.1 Smart Mapping Algorithm

```
# backend/apps/data_migration/services/field_matcher.py
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from fuzzywuzzy import fuzz
import re

class FieldMatcher:
    """
    AI-powered field matching using multiple strategies
    """

    def __init__(self):
        self.vectorizer = TfidfVectorizer()

    def match_fields(self, source_columns, target_schema):
        """
        Match source columns to target schema fields
        Returns list of matches with confidence scores
        """
        matches = []

        for source_col in source_columns:
            best_match = self._find_best_match(
                source_col,
                target_schema
            )
            matches.append(best_match)

        return matches

    def _find_best_match(self, source_col, target_schema):
        """
        Find best matching field using multiple strategies
        """
        candidates = []

        for target_field in target_schema:
            # Strategy 1: Exact match
            exact_score = self._exact_match_score(
                source_col,
                target_field
            )

            # Strategy 2: Fuzzy string matching
            fuzzy_score = self._fuzzy_match_score(
                source_col,
                target_field
            )

            # Strategy 3: Semantic similarity
            semantic_score = self._semantic_similarity(
                source_col,
                target_field
            )

            # Strategy 4: Data type compatibility
            type_score = self._type_compatibility_score(
                source_col,
                target_field
            )

            # Weighted average
            combined_score = (
                exact_score * 0.4 +
                fuzzy_score * 0.25 +
                semantic_score * 0.25 +
                type_score * 0.1
            )

            candidates.append({
                'target_field': target_field['name'],
                'confidence': combined_score,
                'reasons': {
                    'exact': exact_score,
                    'fuzzy': fuzzy_score,
                    'semantic': semantic_score,
                    'type': type_score,
                }
            })

        # Return top match if confidence &gt; threshold
        candidates.sort(key=lambda x: x['confidence'], reverse=True)
        best = candidates[0]

        if best['confidence'] &gt;= 0.6:
            return {
                'source_column': source_col['name'],
                'suggested_field': best['target_field'],
                'confidence': best['confidence'],
                'status': 'auto_matched',
                'alternatives': candidates[1:4]  # Top 3 alternatives
            }
        else:
            return {
                'source_column': source_col['name'],
                'suggested_field': None,
                'confidence': 0,
                'status': 'manual_required',
                'alternatives': candidates[:5]
            }

    def _exact_match_score(self, source, target):
        """Check for exact name match"""
        source_clean = self._normalize_name(source['name'])
        target_clean = self._normalize_name(target['name'])

        if source_clean == target_clean:
            return 1.0

        # Check aliases
        if 'aliases' in target:
            for alias in target['aliases']:
                if source_clean == self._normalize_name(alias):
                    return 0.95

        return 0.0

    def _fuzzy_match_score(self, source, target):
        """Fuzzy string similarity"""
        source_name = source['name'].lower()
        target_name = target['name'].lower()

        # Try multiple fuzzy algorithms
        ratio = fuzz.ratio(source_name, target_name) / 100
        partial = fuzz.partial_ratio(source_name, target_name) / 100
        token_sort = fuzz.token_sort_ratio(source_name, target_name) / 100

        return max(ratio, partial, token_sort)

    def _semantic_similarity(self, source, target):
        """
        Semantic similarity using word embeddings
        For demo: using simple keyword matching
        Production: Use sentence-transformers
        """
        source_keywords = set(self._extract_keywords(source['name']))
        target_keywords = set(self._extract_keywords(target['name']))

        if not source_keywords or not target_keywords:
            return 0.0

        intersection = source_keywords.intersection(target_keywords)
        union = source_keywords.union(target_keywords)

        return len(intersection) / len(union)
    def _type_compatibility_score(self, source, target):
        """Check data type compatibility"""
        source_type = source.get('detected_type', 'string')
        target_type = target.get('type', 'string')

        compatibility_matrix = {
            'integer': ['integer', 'decimal', 'string'],
            'decimal': ['decimal', 'integer', 'string'],
            'string': ['string', 'text'],
            'date': ['date', 'datetime', 'string'],
            'datetime': ['datetime', 'date', 'string'],
            'boolean': ['boolean', 'integer', 'string'],
        }

        if target_type in compatibility_matrix.get(source_type, []):
            return 1.0 if target_type == source_type else 0.7

        return 0.0

    def _normalize_name(self, name):
        """Normalize field name for comparison"""
        # Remove special characters, spaces
        normalized = re.sub(r'[^a-z0-9]', '', name.lower())

        # Common abbreviations
        replacements = {
            'num': 'number',
            'qty': 'quantity',
            'amt': 'amount',
            'addr': 'address',
            'desc': 'description',
            'ref': 'reference',
        }

        for abbr, full in replacements.items():
            normalized = normalized.replace(abbr, full)

        return normalized

    def _extract_keywords(self, name):
        """Extract meaningful keywords from field name"""
        # Split on capital letters, underscores, spaces
        words = re.findall(r'[A-Z]?[a-z]+|[A-Z]+(?=[A-Z][a-z]|\d|\W|$)|\d+', name)

        # Remove common words
        stopwords = {'the', 'a', 'an', 'and', 'or', 'of', 'to', 'in'}
        keywords = [w.lower() for w in words if w.lower() not in stopwords]

        return keywords
```

4.2 Data Profiling Service

```python
# backend/apps/data_migration/services/data_profiler.py
import pandas as pd
import numpy as np
from datetime import datetime

class DataProfiler:
    """
    Analyze source data quality and structure
    """

    def profile_data(self, df, sample_size=100):
        """
        Profile pandas DataFrame
        Returns comprehensive data quality report
        """
        profile = {
            'row_count': len(df),
            'column_count': len(df.columns),
            'columns': [],
            'data_quality': {},
            'sample_rows': [],
        }

        # Profile each column
        for col in df.columns:
            col_profile = self._profile_column(df[col])
            profile['columns'].append(col_profile)

        # Data quality metrics
        profile['data_quality'] = self._assess_quality(df)

        # Sample rows
        sample_df = df.head(min(sample_size, len(df)))
        profile['sample_rows'] = sample_df.to_dict('records')

        return profile

    def _profile_column(self, series):
        """Profile single column"""
        profile = {
            'name': series.name,
            'detected_type': self._detect_type(series),
            'null_count': series.isnull().sum(),
            'null_percent': (series.isnull().sum() / len(series)) * 100,
            'unique_count': series.nunique(),
            'unique_percent': (series.nunique() / len(series)) * 100,
        }

        # Type-specific analysis
        if pd.api.types.is_numeric_dtype(series):
            profile.update(self._profile_numeric(series))
        elif pd.api.types.is_datetime64_any_dtype(series):
            profile.update(self._profile_datetime(series))
        else:
            profile.update(self._profile_text(series))

        # Sample values
        non_null = series.dropna()
        if len(non_null) &gt; 0:
            profile['sample_values'] = non_null.head(5).tolist()

        return profile

    def _detect_type(self, series):
        """Detect column data type"""
        # Try numeric
        try:
            pd.to_numeric(series.dropna(), errors='raise')
            if series.dropna().apply(lambda x: float(x).is_integer()).all():
                return 'integer'
            return 'decimal'
        except (ValueError, TypeError):
            pass

        # Try date/datetime
        try:
            pd.to_datetime(series.dropna(), errors='raise')
            return 'datetime'
        except (ValueError, TypeError):
            pass

        # Try boolean
        unique_vals = series.dropna().unique()
        if len(unique_vals) &lt;= 2:
            bool_values = {'yes', 'no', 'true', 'false', '1', '0', 'y', 'n'}
            if all(str(v).lower() in bool_values for v in unique_vals):
                return 'boolean'

        # Default to string
        return 'string'

    def _profile_numeric(self, series):
        """Profile numeric column"""
        clean = pd.to_numeric(series, errors='coerce').dropna()

        if len(clean) == 0:
            return {}

        return {
            'min': float(clean.min()),
            'max': float(clean.max()),
            'mean': float(clean.mean()),
            'median': float(clean.median()),
            'std': float(clean.std()) if len(clean) &gt; 1 else 0,
        }

    def _profile_datetime(self, series):
        """Profile datetime column"""
        clean = pd.to_datetime(series, errors='coerce').dropna()
        if len(clean) == 0:
            return {}

        return {
            'min_date': clean.min().isoformat(),
            'max_date': clean.max().isoformat(),
            'date_format': self._detect_date_format(series.dropna().iloc[0]),
        }

    def _profile_text(self, series):
        """Profile text column"""
        clean = series.dropna()

        if len(clean) == 0:
            return {}

        lengths = clean.astype(str).str.len()

        return {
            'min_length': int(lengths.min()),
            'max_length': int(lengths.max()),
            'avg_length': float(lengths.mean()),
            'has_special_chars': bool(
                clean.astype(str).str.contains(r'[^a-zA-Z0-9\s]').any()
            ),
        }

    def _assess_quality(self, df):
        """Assess overall data quality"""
        total_cells = df.size
        null_cells = df.isnull().sum().sum()

        # Duplicate rows
        duplicate_count = df.duplicated().sum()

        # Quality score (0-100)
        completeness = ((total_cells - null_cells) / total_cells) * 100
        uniqueness = ((len(df) - duplicate_count) / len(df)) * 100

        quality_score = (completeness * 0.6 + uniqueness * 0.4)

        return {
            'quality_score': round(quality_score, 2),
            'completeness': round(completeness, 2),
            'total_nulls': int(null_cells),
            'null_percentage': round((null_cells / total_cells) * 100, 2),
            'duplicate_rows': int(duplicate_count),
            'duplicate_percentage': round((duplicate_count / len(df)) * 100, 2),
            'issues': self._identify_issues(df),
        }

    def _identify_issues(self, df):
        """Identify data quality issues"""
        issues = []

        # High null columns
        for col in df.columns:
            null_pct = (df[col].isnull().sum() / len(df)) * 100
            if null_pct &gt; 50:
                issues.append({
                    'type': 'high_nulls',
                    'column': col,
                    'severity': 'warning',
                    'message': f"Column '{col}' has {null_pct:.1f}% null values"
                })

        # Duplicate rows
        if df.duplicated().sum() &gt; 0:
            issues.append({
                'type': 'duplicates',
                'severity': 'warning',
                'message': f"Found {df.duplicated().sum()} duplicate rows"
            })

        return issues

    def _detect_date_format(self, sample):
        """Detect date format from sample"""
        formats = [
            '%Y-%m-%d', '%d-%m-%Y', '%m-%d-%Y',
            '%Y/%m/%d', '%d/%m/%Y', '%m/%d/%Y',
            '%d.%m.%Y', '%Y.%m.%d',
        ]

        for fmt in formats:
            try:
                datetime.strptime(str(sample), fmt)
                return fmt
            except ValueError:
                continue

        return 'unknown'
```

5. Data Validation Engine

```
# backend/apps/data_migration/services/validator.py
from pydantic import BaseModel, validator, ValidationError
from typing import Any, Dict, List
import re

class ValidationRule(BaseModel):
    """Validation rule definition"""
    field: str
    rule_type: str  # required, unique, format, range, etc.
    parameters: Dict[str, Any] = {}
    error_message: str = ""

class DataValidator:
    """
    Validate data before import
    """

    def __init__(self, target_model, custom_rules=None):
        self.target_model = target_model
        self.custom_rules = custom_rules or []

    def validate_batch(self, data_rows):
        """
        Validate batch of rows
        Returns validation results with errors
        """
        results = {
            'valid_count': 0,
            'error_count': 0,
            'rows': []
        }

        for idx, row in enumerate(data_rows):
            row_result = self.validate_row(row, idx + 1)
            results['rows'].append(row_result)

            if row_result['is_valid']:
                results['valid_count'] += 1
            else:
                results['error_count'] += 1

        return results

    def validate_row(self, row_data, row_number):
        """Validate single row"""
        errors = []
        warnings = []

        # Required field validation
        errors.extend(self._validate_required_fields(row_data))

        # Data type validation
        errors.extend(self._validate_types(row_data))

        # Format validation
        errors.extend(self._validate_formats(row_data))

        # Range validation
        errors.extend(self._validate_ranges(row_data))

        # Custom business rules
        errors.extend(self._validate_custom_rules(row_data))

        # Data quality warnings
        warnings.extend(self._check_data_quality(row_data))

        return {
            'row_number': row_number,
            'is_valid': len(errors) == 0,
            'errors': errors,
            'warnings': warnings,
            'data': row_data,
        }

    def _validate_required_fields(self, row_data):
        """Check required fields"""
        errors = []
        required_fields = self._get_required_fields()

        for field in required_fields:
            if field not in row_data or row_data[field] in [None, '', 'null']:
                errors.append({
                    'field': field,
                    'type': 'required',
                    'message': f"Field '{field}' is required"
                })

        return errors

    def _validate_types(self, row_data):
        """Validate data types"""
        errors = []
        field_types = self._get_field_types()

        for field, expected_type in field_types.items():
            if field not in row_data or row_data[field] is None:
                continue

            value = row_data[field]

            try:
                if expected_type == 'integer':
                    int(value)
                elif expected_type == 'decimal':
                    float(value)
                elif expected_type == 'date':
                    pd.to_datetime(value)
                elif expected_type == 'email':
                    if not re.match(r'^[\w\.-]+@[\w\.-]+\.\w+$', value):
                        raise ValueError("Invalid email")
            except (ValueError, TypeError):
                errors.append({
                    'field': field,
                    'type': 'type_error',
                    'message': f"Field '{field}' must be {expected_type}"
                })

        return errors

    def _validate_formats(self, row_data):
        """Validate data formats"""
        errors = []
        format_rules = {
            'email': r'^[\w\.-]+@[\w\.-]+\.\w+$',
            'phone': r'^\+?[\d\s\-\(\)]+$',
            'tax_id': r'^[\w\-]+$',
        }
        for field, pattern in format_rules.items():
            if field in row_data and row_data[field]:
                if not re.match(pattern, str(row_data[field])):
                    errors.append({
                        'field': field,
                        'type': 'format_error',
                        'message': f"Invalid format for '{field}'"
                    })

        return errors

    def _validate_ranges(self, row_data):
        """Validate numeric ranges"""
        errors = []

        # Example: price must be positive
        if 'price' in row_data:
            try:
                price = float(row_data['price'])
                if price &lt; 0:
                    errors.append({
                        'field': 'price',
                        'type': 'range_error',
                        'message': 'Price must be positive'
                    })
            except (ValueError, TypeError):
                pass

        return errors

    def _validate_custom_rules(self, row_data):
        """Validate custom business rules"""
        errors = []

        for rule in self.custom_rules:
            if not self._check_rule(row_data, rule):
                errors.append({
                    'field': rule.field,
                    'type': 'business_rule',
                    'message': rule.error_message or f"Failed rule: {rule.rule_type}"
                })

        return errors

    def _check_data_quality(self, row_data):
        """Check data quality issues (non-blocking)"""
        warnings = []

        # Check for suspicious values
        for field, value in row_data.items():
            if isinstance(value, str):
                # Check for placeholder values
                placeholders = ['n/a', 'na', 'null', 'none', 'tbd', 'xxx']
                if value.lower() in placeholders:
                    warnings.append({
                        'field': field,
                        'type': 'suspicious_value',
                        'message': f"Suspicious placeholder value: '{value}'"
                    })

        return warnings

    def _get_required_fields(self):
        """Get required fields for target model"""
        # This would be loaded from model schema
        return ['name', 'code']

    def _get_field_types(self):
        """Get field types for target model"""
        # This would be loaded from model schema
        return {
            'code': 'string',
            'quantity': 'decimal',
            'price': 'decimal',
            'date': 'date',
        }

    def _check_rule(self, row_data, rule):
        """Check individual validation rule"""
        # Simplified rule checking
        if rule.rule_type == 'unique':
            # Would check against existing data
            return True
        elif rule.rule_type == 'reference':
            # Would check foreign key references
            return True

        return True
```

6. Data Transformation Service

```
# backend/apps/data_migration/services/transformer.py
import pandas as pd
import re
from datetime import datetime

class DataTransformer:
    """
    Transform and cleanse data before import
    """

    def transform_batch(self, rows, transformation_rules):
        """Apply transformations to batch"""
        transformed = []

        for row in rows:
            transformed_row = self.transform_row(row, transformation_rules)
            transformed.append(transformed_row)
        return transformed

    def transform_row(self, row_data, rules):
        """Transform single row"""
        transformed = row_data.copy()

        for field, rule_list in rules.items():
            if field not in transformed:
                continue

            value = transformed[field]

            for rule in rule_list:
                value = self._apply_transformation(value, rule)

            transformed[field] = value

        return transformed

    def _apply_transformation(self, value, rule):
        """Apply single transformation rule"""
        rule_type = rule.get('type')

        if value is None or value == '':
            # Handle default values
            if rule_type == 'default':
                return rule.get('value')
            return value

        if rule_type == 'trim':
            return str(value).strip()

        elif rule_type == 'uppercase':
            return str(value).upper()

        elif rule_type == 'lowercase':
            return str(value).lower()

        elif rule_type == 'title_case':
            return str(value).title()

        elif rule_type == 'remove_special_chars':
            return re.sub(r'[^a-zA-Z0-9\s]', '', str(value))

        elif rule_type == 'normalize_phone':
            return self._normalize_phone(value)

        elif rule_type == 'date_format':
            return self._convert_date_format(value, rule.get('format'))

        elif rule_type == 'replace':
            return str(value).replace(
                rule.get('find', ''),
                rule.get('replace', '')
            )
        elif rule_type == 'mapping':
            mappings = rule.get('mappings', {})
            return mappings.get(str(value), value)

        elif rule_type == 'calculate':
            return self._calculate_value(value, rule.get('formula'))

        return value

    def _normalize_phone(self, phone):
        """Normalize phone number"""
        # Remove all non-numeric
        digits = re.sub(r'\D', '', str(phone))

        # Format based on length
        if len(digits) == 11 and digits.startswith('0'):
            # Bangladesh mobile: 01XXXXXXXXX
            return f"+88{digits}"
        elif len(digits) == 10:
            return f"+88{digits}"

        return phone

    def _convert_date_format(self, date_value, target_format='%Y-%m-%d'):
        """Convert date to standard format"""
        try:
            parsed = pd.to_datetime(date_value)
            return parsed.strftime(target_format)
        except:
            return date_value

    def _calculate_value(self, value, formula):
        """Calculate derived value"""
        # Simple formula evaluation
        # Production: Use safe expression evaluator
        try:
            return eval(formula.replace('x', str(value)))
        except:
            return value
```

7. Import Engine

```
# backend/apps/data_migration/services/importer.py
from django.db import transaction
from django.apps import apps

class DataImporter:
    """
    Import validated and transformed data
    """

    def __init__(self, session):
        self.session = session
        self.model = self._get_target_model()
    def import_data(self, validated_rows):
        """
        Import data with transaction support
        """
        imported_count = 0
        error_count = 0

        try:
            with transaction.atomic():
                for row_result in validated_rows['rows']:
                    if not row_result['is_valid']:
                        error_count += 1
                        self._log_error(row_result)
                        continue

                    try:
                        record = self._import_row(row_result['data'])
                        self._track_imported_record(
                            row_result['row_number'],
                            record,
                            row_result['data']
                        )
                        imported_count += 1
                    except Exception as e:
                        error_count += 1
                        self._log_error(row_result, str(e))

                # Update session stats
                self.session.success_rows = imported_count
                self.session.error_rows = error_count
                self.session.status = 'COMPLETED'
                self.session.save()

        except Exception as e:
            self.session.status = 'FAILED'
            self.session.save()
            raise

        return {
            'imported': imported_count,
            'errors': error_count,
        }

    def _import_row(self, row_data):
        """Import single row"""
        # Add company context
        row_data['company'] = self.session.company

        # Handle foreign keys
        row_data = self._resolve_foreign_keys(row_data)

        # Create record
        record = self.model.objects.create(**row_data)

        return record
    def _resolve_foreign_keys(self, row_data):
        """Resolve foreign key references"""
        # Example: Resolve account by code
        if 'account_code' in row_data:
            from apps.finance.models import Account
            account = Account.objects.get(
                company=self.session.company,
                code=row_data['account_code']
            )
            row_data['account'] = account
            del row_data['account_code']

        return row_data

    def _get_target_model(self):
        """Get Django model for import"""
        return apps.get_model(
            self.session.target_module,
            self.session.target_model
        )

    def _log_error(self, row_result, exception=None):
        """Log import error"""
        from apps.data_migration.models import MigrationError

        MigrationError.objects.create(
            session=self.session,
            row_number=row_result['row_number'],
            source_data=row_result['data'],
            error_type='import_error',
            error_message=exception or row_result['errors'][0]['message']
        )

    def _track_imported_record(self, row_number, record, source_data):
        """Track imported record for rollback"""
        from apps.data_migration.models import MigrationRecord

        MigrationRecord.objects.create(
            session=self.session,
            target_model=self.session.target_model,
            target_id=record.id,
            source_row_number=row_number,
            source_data=source_data
        )
```

8. Rollback Service

```
# backend/apps/data_migration/services/rollback.py
from django.db import transaction
from django.apps import apps

class RollbackService:
    """
    Rollback imported data
    """

    def rollback_session(self, session):
        """
        Rollback all records from a migration session
        """
        if session.status not in ['COMPLETED', 'FAILED']:
            raise ValueError("Can only rollback completed or failed sessions")

        deleted_count = 0

        try:
            with transaction.atomic():
                # Get all imported records
                records = session.records.all()

                for record in records:
                    try:
                        # Get the actual model instance
                        model = apps.get_model(
                            session.target_module,
                            record.target_model
                        )
                        instance = model.objects.get(id=record.target_id)
                        instance.delete()
                        deleted_count += 1
                    except model.DoesNotExist:
                        # Already deleted
                        pass

                # Mark session as rolled back
                session.status = 'ROLLED_BACK'
                session.save()

        except Exception as e:
            raise Exception(f"Rollback failed: {str(e)}")

        return deleted_count
```

9. Implementation Checklist

Weeks 1-2: Foundation

- ☐ Create migration models (Session, Template, Profile)
- ☐ Implement file upload handling
- ☐ Build data profiling service
- ☐ Write unit tests

Weeks 3-4: AI Matching

- ☐ Implement field matching algorithm
- ☐ Train/test on sample datasets
- ☐ Build confidence scoring
- ☐ Create alternative suggestions

Weeks 5-6: Validation & Transformation

- ☐ Build validation engine
- ☐ Implement transformation rules
- ☐ Create data cleansing functions
- ☐ Write integration tests

Weeks 7-8: Import & UI

- ☐ Implement import engine
- ☐ Build rollback service
- ☐ Create drag-and-drop mapping UI
- ☐ Build progress tracking UI
- ☐ End-to-end testing

Document Control:

- **Version:** 1.0
- **Dependencies:** Phase 0, 1, 2 complete
- **Next Phase:** Phase 4 - No-Code Builders