Phase 1C Part 3: Data Validation and Transformation Layer

Overview

This document describes the comprehensive data validation and transformation layer implemented for the Municipal Transparency Platform. This layer ensures data quality and consistency across all data ingestion points (file uploads and API connectors).

Architecture

Components

1. **Validation Service** (`apps/api/src/validation/`)
   - Comprehensive validation rules for all entity types
   - Field-level validation with detailed error reporting
   - Foreign key relationship validation
   - Data type and range validation

2. **Transformation Service** (`apps/api/src/transformation/`)
   - Data normalization from various sources
   - Date format conversion
   - Currency value normalization
   - Text field standardization
   - Status code mapping

3. **Custom Exceptions** (`apps/api/src/validation/exceptions/`)
   - Specialized exception classes for different validation failure types
   - Structured error responses with field-level details
   - Helpful suggestions for fixing validation errors

Validation Service

Supported Entities

1. Budget Validation

- **Required Fields**: fiscalYear, department, program, category, subcategory, amountPlanned
- **Validations**:
- Fiscal year must be between 2000 and current year + 10
- Amount must be positive and within reasonable limits (< 999,999,999,999.99)
- String fields have length constraints (max 255 characters)
- Currency must be valid (CLP, UF, USD, EUR)
- Municipality ID foreign key validation (optional)

2. Expenditure Validation

- **Required Fields**: date, fiscalYear, department, program, category, subcategory, concept, amountActual

- **Validations**:
- Date must be valid and within acceptable range (2000-01-01 to 1 year in future)
- Fiscal year validation
- Amount validation (positive, within limits)
- String field length constraints
- Currency validation
- Supplier ID foreign key validation (optional)

3. Project Validation

- **Required Fields**: title, description, status, department, category
- **Validations**:
- Title: 1-500 characters
- Description: minimum 10 characters
- Status: must be one of [planificado, en_progreso, completado, cancelado, suspendido]
- Start date must be before end date
- Budget amounts must be non-negative
- Funding source foreign key validation (optional)

4. Contract Validation

- **Required Fields**: title, description, amount, startDate, status, supplierId
- **Validations**:
- Title: 1-500 characters
- Description: minimum 10 characters
- Amount: positive and within limits
- Status: must be one of [activo, completado, cancelado, suspendido, en_revision]
- Start date must be before end date
- Contract number uniqueness
- Supplier foreign key validation

Validation Methods

```
// Single record validation
async validateBudget(data: any, context?: IValidationContext): Promise<IValidationRes-
ult>
async validateExpenditure(data: any, context?: IValidationContext): Promise<IValida-
tionResult>
async validateProject(data: any, context?: IValidationContext): Promise<IValidation-
Result>
async validateContract(data: any, context?: IValidationContext): Promise<IValidation-
Result>

// Batch validation
async validateBatch(
  entityType: 'budget' | 'expenditure' | 'project' | 'contract',
  records: any[],
  context?: IValidationContext
): Promise<{ validRecords: any[]; invalidRecords: any[] }>
```

Validation Context

```
interface IValidationContext {
  entityType: 'budget' | 'expenditure' | 'project' | 'contract';
  municipalityId?: string;
  skipForeignKeyValidation?: boolean;
  strictMode?: boolean;
}
```

Transformation Service

Features

1. **Date Normalization**
   - Supports multiple input formats:

   - ISO 8601 (YYYY-MM-DD, YYYY-MM-DDTHH:mm:ss.sssZ)
   - DD/MM/YYYY (Chilean standard)
   - DD-MM-YY
   - Converts all dates to ISO format

2. **Currency Normalization**
   - Handles different decimal separators (, or .)
   - Removes thousands separators
   - Removes currency symbols ($, CLP, UF, etc.)
   - Validates and converts to numeric values

3. **Currency Code Mapping**
   - Maps common variations to standard codes
   - Examples: “Peso” → “CLP”, “Unidad de Fomento” → “UF”, “Dólar” → “USD”

4. **Status Normalization**
   - **Project Status**:

   - “planned”, “planeado” → “planificado”
   - “in progress”, “en curso”, “activo” → “en_progreso”
   - “complete”, “completed”, “finalizado” → “completado”
   - “cancelled”, “canceled” → “cancelado”
   - **Contract Status**:
   - “active”, “vigente” → “activo”
   - “complete”, “finalizado” → “completado”
   - “review”, “en revisión” → “en_revision”

5. **Text Normalization**
   - Trims whitespace
   - Replaces multiple spaces with single space
   - Removes leading/trailing whitespace

6. **Field Mapping**
   - Supports custom field name mappings
   - Allows custom transformation functions
   - Handles default values for missing fields

Transformation Methods

```
// Single record transformation
transformBudget(data: any, config?: IDataSourceConfig): ITransformationResult
transformExpenditure(data: any, config?: IDataSourceConfig): ITransformationResult
transformProject(data: any, config?: IDataSourceConfig): ITransformationResult
transformContract(data: any, config?: IDataSourceConfig): ITransformationResult

// Batch transformation
transformBatch(
  entityType: 'budget' | 'expenditure' | 'project' | 'contract',
  records: any[],
  config?: IDataSourceConfig
): ITransformationResult[]
```

Data Source Configuration

```
interface IDataSourceConfig {
  source: 'csv' | 'excel' | 'json' | 'api';
  fieldMappings?: IFieldMapping[];
  dateFormat?: string;
  currencyFormat?: string;
  decimalSeparator?: '.' | ',';
  thousandsSeparator?: ',' | '.' | ' ' | '';
}
```

Custom Exceptions

Exception Types

1. **ValidationException**: Base exception for validation errors
2. **RequiredFieldException**: For missing required fields
3. **TypeValidationException**: For data type mismatches
4. **RangeValidationException**: For values outside allowed ranges
5. **ForeignKeyValidationException**: For invalid foreign key references
6. **EnumValidationException**: For invalid enum values
7. **DateValidationException**: For invalid date formats or values
8. **CurrencyValidationException**: For invalid currency values

Error Response Format

```
{
  statusCode: 400,
  message: "Validation failed",
  errors: [
    {
      field: "amountPlanned",
      value: "-1000",
      message: "El monto planificado debe ser mayor que cero",
      constraint: "range",
      suggestion: "Proporcione un monto positivo"
    }
  ],
  timestamp: "2025-10-16T10:30:00.000Z"
}
```

Integration

File Upload Integration

The validation and transformation services are integrated into the upload module:

```
// apps/api/src/upload/upload.module.ts
@Module({
  imports: [PrismaModule, ValidationModule, TransformationModule],
  // ...
})
export class UploadModule {}
```

The UploadService now has access to both services through dependency injection:

```
constructor(
  private prisma: PrismaService,
  private validationService: ValidationService,
  private transformationService: TransformationService,
) {}
```

API Connector Integration

The connectors module also includes validation and transformation:

```
// apps/api/src/connectors/connectors.module.ts
@Module({
  imports: [
    HttpModule,
    PrismaModule,
    ValidationModule,
    TransformationModule,
  ],
  // ...
})
export class ConnectorsModule {}
```

Connector implementations (BudgetSourceConnector, ChileCompraConnector) receive these services:

```
constructor(
  config: IConnectorConfig,
  httpService: HttpService,
  logService: ConnectorLogService,
  private readonly prisma: PrismaService,
  private readonly validationService?: ValidationService,
  private readonly transformationService?: TransformationService,
) {
  super(config, httpService, logService);
}
```

Usage Examples

Example 1: Validating Budget Data

```
const budgetData = {
  fiscalYear: 2024,
  department: 'Educación',
  program: 'Programa Educativo',
  category: 'Personal',
  subcategory: 'Docentes',
  amountPlanned: 1000000,
  currency: 'CLP',
};

const result = await validationService.validateBudget(budgetData, {
  entityType: 'budget',
  municipalityId: 'municipality-id-123',
});

if (result.isValid) {
  // Save to database
} else {
  // Handle errors
  console.log(result.errors);
}
```

Example 2: Transforming Expenditure Data

```javascript
const rawData = {
  date: '15/10/2024', // Chilean format
  fiscalYear: '2024',
  department: '  Salud  ', // With whitespace
  program: 'Programa de Salud',
  category: 'Bienes y Servicios',
  subcategory: 'Medicamentos',
  concept: 'Compra de medicamentos',
  amountActual: '$1.500.000,50', // Chilean currency format
  currency: 'Peso Chileno',
};

const transformed = transformationService.transformExpenditure(rawData, {
  source: 'csv',
  decimalSeparator: ',',
  thousandsSeparator: '.',
});

// transformed.data:
// {
//   date: Date('2024-10-15T00:00:00.000Z'),
//   fiscalYear: 2024,
//   department: 'Salud',
//   program: 'Programa de Salud',
//   category: 'Bienes y Servicios',
//   subcategory: 'Medicamentos',
//   concept: 'Compra de medicamentos',
//   amountActual: 1500000.50,
//   currency: 'CLP',
// }
```

Example 3: Batch Processing

```
const records = [
  { /* budget data 1 */ },
  { /* budget data 2 */ },
  { /* budget data 3 */ },
];

// Transform all records
const transformedResults = transformationService.transformBatch('budget', records);

// Extract transformed data
const transformedData = transformedResults.map(r => r.data);

// Validate all records
const { validRecords, invalidRecords } = await validationService.validateBatch(
  'budget',
  transformedData,
  { entityType: 'budget', municipalityId: 'muni-123' }
);

// Process valid records
for (const record of validRecords) {
  // Save to database
}

// Report invalid records
for (const record of invalidRecords) {
  console.log('Row', record.rowNumber, 'errors:', record.validationErrors);
}
```

Error Handling Best Practices

1. **Always catch validation exceptions**:
   ```
   typescript
      try {
        const result = await validationService.validateBudget(data);
        // ...
      } catch (error) {
        if (error instanceof ValidationException) {
          // Handle validation errors
          console.log(error.getFormattedMessage());
          console.log(error.getErrorsByField());
        }
      }
   ```

2. **Provide user-friendly error messages**:
   - Use the `suggestion` field to guide users
   - Group errors by field for better UX
   - Include the original value in error responses

3. **Log validation warnings from transformation**:
   ```
   typescript
      const result = transformationService.transformBudget(data);
      if (result.warnings && result.warnings.length > 0) {
   ```

```
    logger.warn('Transformation warnings:', result.warnings);
  }
```

Testing

To test the validation and transformation services:

```
# Run all tests
npm test

# Run specific test file
npm test validation.service.spec.ts
npm test transformation.service.spec.ts

# Run with coverage
npm test -- --coverage
```

Performance Considerations

1. **Batch Operations**: Use batch methods for processing multiple records to reduce database queries
2. **Skip Foreign Key Validation**: Set `skipForeignKeyValidation: true` when importing large datasets to improve performance
3. **Caching**: Consider caching frequently validated patterns or reference data
4. **Async Processing**: For large imports, consider using background jobs

Future Enhancements

1. **Custom Validation Rules**: Allow configuration of custom validation rules per municipality
2. **Configurable Transformations**: Support for user-defined transformation rules
3. **Machine Learning**: Implement ML-based data quality suggestions
4. **Real-time Validation**: WebSocket-based real-time validation for large file uploads
5. **Validation Templates**: Pre-configured validation sets for common data sources

Related Documentation

- Phase 1C Part 1: File Upload Functionality (./PHASE_1C_PART_1_FILE_UPLOAD.md)
- Phase 1C Part 2: API Connectors (./PHASE_1C_PART_2_API_CONNECTORS.md)
- Database Schema (../packages/database/prisma/schema.prisma)

Support

For questions or issues:
- Create an issue in the project repository
- Contact the development team
- Refer to the inline code documentation