Submit

2025-09-24T10:56:09+07:00

Contents

Submit Pre-Requisition - Detailed Design Document

1. Introduction

1.1. Purpose

This document provides a detailed design specification for the Submit Pre-Requisition functionality in the MIMS (Medical Inventory Management System) backend application. The document outlines the system architecture, component interactions, data flow, and implementation details for processing pre-requisition submissions.

1.2. Scope

This document covers: - Pre-requisition submission workflow - User authorization and validation - Pre-requisition processing and separation logic - X12 file generation for external system integration - Error handling and logging mechanisms - Database interactions and transaction management

1.3. Definitions, Acronyms, and Abbreviations

Term	Definition
MIMS	Medical Inventory Management System
Pre-Requisition	A request for medical supplies/equipment before a surgical procedure
X12	EDI (Electronic Data Interchange) standard for business transactions
MSS	Master Supply System - external item master database
UOM	Unit of Measure
QOE	Quantity Per
2FA	Two-Factor Authentication
EDI	Electronic Data Interchange
ERP	Enterprise Resource Planning
API	Application Programming Interface

1.4. References

- Pre-Requisition Submit Service Code
- X12ProcessImproved Implementation
- Pre-Requisition Helper
- Sub-Item Separation Workflow Documentation
- X12 File Creation Workflow Documentation

2. Component Responsibilities

2.1. PreRequisition::Submit Service

- **Primary Responsibility**: Orchestrate the complete pre-requisition submission process
- **Key Functions**:
  - Validate user permissions and pre-requisition data
  - Process pre-requisition separations
  - Generate X12 files for external integration
  - Manage database transactions
  - Handle error conditions and logging

2.2. X12ProcessImproved Class

- **Primary Responsibility**: Generate standardized EDI X12 files
- **Key Functions**:
  - Format data according to X12 850 (Purchase Order) standard
  - Handle character encoding and field validation
  - Manage file output and permissions
  - Process item, inventory, and comment data

2.3. PreRequisitionHelper Module

- **Primary Responsibility**: Provide utility methods and data preparation
- **Key Functions**:
  - Build data structures for X12 generation
  - Generate file names and paths
  - Handle organization settings and configurations
  - Manage MSS database connections

2.4. ApplicationLogger

- **Primary Responsibility**: Provide structured logging for audit and debugging
- **Key Functions**:
  - Log submission start/completion events
  - Record separation activities
  - Track file creation processes
  - Monitor error conditions

3. Constraints

3.1. Technical Constraints

- **Database Transactions**: All pre-requisition updates must be atomic
- **File System**: X12 files must be written to `ENV['INTEGRATION_PATH']` directory

- **Memory Usage**: Large pre-requisitions must be processed efficiently to avoid memory issues
- **Concurrent Access**: System must handle multiple simultaneous submissions

3.2. Business Constraints

- **User Authorization**: Only authorized users can submit pre-requisitions
- **Status Validation**: Only pre-requisitions in valid states can be submitted
- **Item Validation**: All items must have valid quantities and pricing
- **Attachment Requirements**: Some organizations require attachments for certain items

3.3. Integration Constraints

- **X12 Format**: Files must conform to EDI 850 standard
- **File Naming**: Files must follow specific naming convention for external systems
- **Character Encoding**: Special characters must be properly encoded
- **File Permissions**: Files must have correct permissions for external system access

4. Uses/Interactions

4.1. Uses

4.1.1. External Dependencies

- **MSS Database**: For item master data and pricing information
- **File System**: For X12 file storage and management
- **Organization Settings**: For configuration and business rules
- **User Management**: For authentication and authorization

4.1.2. Internal Dependencies

- **Pre-Requisition Models**: Core business entities
- **Inventory Models**: Item inventory tracking
- **Action Logging**: Audit trail management
- **Helper Modules**: Utility functions and data preparation

4.2. Interactions

4.2.1. User Interface Interactions

- **API Endpoints**: RESTful API for submission requests
- **Authentication**: User login and session management
- **Authorization**: Permission checking and validation

- **Response Handling**: Success/error message delivery

4.2.2. External System Interactions

- **X12 File Generation**: For ERP system integration
- **File Transfer**: Automated file delivery to external systems
- **Status Updates**: Real-time submission status tracking
- **Error Notifications**: Failure alert mechanisms

5. Resources (Optional)

5.1. Database Resources

- **Primary Database**: PostgreSQL for main application data
- **MSS Database**: External item master database
- **Connection Pooling**: For efficient database access
- **Transaction Management**: ACID compliance for data integrity

5.2. File System Resources

- **Integration Directory**: `ENV['INTEGRATION_PATH']` for X12 files
- **File Permissions**: 0o664 for external system access
- **Disk Space**: Adequate storage for file generation
- **Backup Systems**: File retention and archival

5.3. System Resources

- **Memory**: Sufficient RAM for large pre-requisition processing
- **CPU**: Processing power for concurrent submissions
- **Network**: Bandwidth for external system communication
- **Monitoring**: System health and performance tracking

6. Detailed Designs

6.1. Back-End Design

6.1.1. Overview Process Submit Pre-Requisition The submit pre-requisition process follows a structured workflow designed to ensure data integrity, proper validation, and successful integration with external systems.

High-Level Process Flow:

The submit pre-requisition process follows a structured workflow with multiple validation steps, data processing, and external integrations. The following sequence diagram illustrates the overview of interactions between components:

Process Overview: - **Initialization**: User request validation and service setup - **Authorization**: User and pre-requisition validation - **Data Processing**: MSS connection and item processing - **Separation Logic**: Pre-requisition

Submit Pre-Requisition Overview Sequence

Figure 1: Submit Pre-Requisition Overview Sequence

and sub-item separation - **File Generation**: X12 file creation for external systems - **Completion**: Status updates and logging

Key Process Steps:

1. **Initialization**: Set up logging, parameters, and user context
2. **Validation**: Verify user permissions and pre-requisition status
3. **Data Processing**: Connect to MSS, validate items, and process inventories
4. **Separation Logic**: Apply business rules for pre-requisition separation
5. **File Generation**: Create X12 files for external system integration
6. **Completion**: Update status, log results, and return response

Process Entry Point:

```
def call
  org_id = @params['org_id'].to_i
  @log_actions = {}
  @created_pre_req_items = []

  status, message, focus_id = process_submit

  return [status, message] if status == ERROR_STATUS

  write_created_updated_log(@log_actions)
  [status, message, focus_id]
end
```

6.1.2. Authorize User User authorization is the first critical step in the submission process, ensuring only authorized users can submit pre-requisitions.

Authorization Components:

1. **User Validation**:
   - Verify user exists and is active
   - Check user permissions for the organization
   - Validate user role and access levels
2. **Pre-Requisition Access**:
   - Confirm user can access the specific pre-requisition
   - Verify organization membership
   - Check surgical case permissions

Authorization Logic:

```
def process_submit
```

```
  return failure if @current_user.blank?

  pre_req_id = @params['pre_requisition_id'].to_i
  pre_requisition = Org::PreRequisition.find_by_id(pre_req_id)

  return failure unless pre_requisition_valid?(pre_requisition)

  # Additional authorization checks...
end
```

Validation Methods: - `pre_requisition_valid?`: Checks pre-requisition existence and status - `surgical_case_valid?`: Validates surgical case permissions - `ship_to_loc_valid?`: Verifies location access rights

Error Handling: - Returns 422 status for authorization failures - Logs unauthorized access attempts - Provides appropriate error messages to users

6.1.3. Process Submit Pre-Requisition

6.1.3.1. Overview The core submission process handles the complete workflow from validation to completion, including all business logic and external integrations.

Process Architecture:

```
def process_submit
  # 1. Initial validation and setup
  item_status = true
  return failure if @current_user.blank?

  # 2. Data retrieval and validation
  pre_req_id = @params['pre_requisition_id'].to_i
  pre_requisition = Org::PreRequisition.find_by_id(pre_req_id)

  # 3. MSS connection and item processing
  surgical_case = Org::SurgicalCase.find_by_id(pre_requisition.org_surgical_case_id)
  org_id = surgical_case.org_id
  connect_mss(org_id)

  # 4. Item validation and correction
  pre_req_items = Org::PreRequisitionItem.where(org_pre_requisition_id: pre_req_id)
  pre_req_items.each do |item|
    item_status = correct_item_info(item) if item.mss_source
    break unless item_status
  end

  # 5. Separation processing
```

```
  if @re_submit
    merge_item(pre_requisition)
    move_inventory(pre_requisition, pre_requisition, status: WASTED_STATUS)
    pre_req_ids, pre_req_numbers = clear_blank_pre_req([pre_requisition])
  else
    pre_req_ids, pre_req_numbers = separate_pre_req(pre_requisition)
  end

  # 6. Sub-item separation
  if should_separate_by_sub_items?(org_id)
    pre_req_ids, pre_req_numbers = separate_by_sub_items(pre_req_ids)
  end

  # 7. File generation and completion
  write_log_create_pre_req_item
  create_line_nums(pre_req_ids)
  file_paths = create_files(pre_req_ids)
  update_permission_to_files(file_paths)
  update_pre_req_info(pre_requisition, pre_req_stt)
  update_submitted_pre_reqs(pre_req_ids)

  success(build_message(pre_req_numbers)) << pre_req_ids.first
end
```

6.1.3.2. Validate User, Pre-Requisition Comprehensive validation ensures data integrity and business rule compliance before processing.

User Validation:

```
def pre_requisition_valid?(pre_requisition)
  return false unless pre_requisition.present?
  return false if PRE_REQ_SYNC_STT.include?(pre_requisition.pre_req_status)
  true
end
```

Pre-Requisition Validation: - **Existence Check**: Verify pre-requisition exists - **Status Validation**: Ensure pre-requisition is in submittable state - **Surgical Case Validation**: Confirm surgical case is valid and not overdue - **Location Validation**: Verify ship-to location requirements

Item Validation:

```
def pre_requisition_items_valid?(pre_req_items)
  return false if pre_req_items.blank?
  pre_req_items.positive_quantity.none? { |item| item.status != APPROVED }
end
```

Inventory Validation: - **Quantity Validation**: Ensure all items have pos-

itive quantities - **Status Validation**: Verify all items are approved - **Attachment Validation**: Check required attachments based on org settings - **Lot/Serial/Expiration**: Validate required tracking information

Error Handling: - Returns specific error messages for each validation failure - Logs validation errors for debugging - Provides user-friendly error descriptions

6.1.3.3. Connect MSS MSS (Master Supply System) connection is essential for item validation and pricing updates.

Connection Process:

```
def connect_mss(org_id)
  # Establish connection to MSS database
  # Configure connection parameters
  # Verify connection status
end
```

MSS Integration Functions: - **Item Validation**: Verify items exist in MSS - **Price Updates**: Update item pricing from MSS - **Manufacturer Data**: Sync manufacturer information - **Catalog Updates**: Update item catalog numbers

Item Correction Process:

```
def correct_item_info(pre_req_item)
  correct_item_price(pre_req_item)
  pre_req_item.calculate_costs
  pre_req_item.calculate_item_quantity

  mss_item = Org::Mss::MasterItem.find_by_id(pre_req_item.vendor_item_id)
  return false unless mss_item.present?

  pre_req_item.mfr_name = mss_item.approved_mfr_name
  pre_req_item.mfr_code = mss_item.mfr_id
  pre_req_item.mfr_cat_num = mss_item.approved_mfr_item_id
  pre_req_item.save
end
```

Error Handling: - Handles MSS connection failures gracefully - Logs MSS integration errors - Continues processing with available data

6.1.3.4. Separate Pre-Requisition Pre-requisition separation applies business rules to split requisitions based on organizational settings.

Separation Logic:

```
def separate_pre_req(pre_req)
  pre_req_list = []
  @org_filter.each_with_index do |condition, index|
    current_condition = condition
```

```
    if index.zero?
      # Update original pre-requisition
      old_type = pre_req.pre_requisition_type
      pre_req.update(pre_requisition_type: convert_condition_to_type(condition))
      new_pre_req = pre_req
      current_condition = { status: WASTED_STATUS }
    else
      # Create new pre-requisition
      new_pre_req = clone_pre_req(pre_req, condition)
    end

    move_inventory(pre_req, new_pre_req, current_condition)
    process_separate_construct(new_pre_req)
    update_info(new_pre_req, condition)
    pre_req_list << new_pre_req
  end
  clear_blank_pre_req(pre_req_list)
end
```

Separation Criteria: - **Status Separation**: Normal vs. wasted items - **Bill Type Separation**: Bill only vs. bill replace - **Organization Settings**: Configurable separation rules

Inventory Movement:

```
def move_inventory(pre_req, new_pre_req, condition)
  items = Org::PreRequisitionItem.where(org_pre_requisition_id: pre_req.id)
  inventories = Org::PreRequisitionInventory.where(org_pre_requisition_item_id: items.pluck(
                                             .filter_for_separate(condition)

  inventories.each do |inventory|
    existed_item = items.find { |item| item.id == inventory.org_pre_requisition_item_id }
    new_item_id = find_or_create_item_for_inventory(existed_item, new_pre_req, inventory)
    inventory.update(org_pre_requisition_item_id: new_item_id)
    # Additional processing...
  end
end
```

6.1.3.5. Separate by Setting Sub-item separation is an advanced feature that separates items based on inventory identifiers.

Separation Condition Check:

```
def should_separate_by_sub_items?(org_id)
  org_type = detect_type_for_org(org_id)
  separation_enabled = sub_item_separation_enabled?(org_id)

  @logger.info({
```

```
    action: 'checking_separation_conditions',
    org_id: org_id,
    org_type: org_type,
    separation_enabled: separation_enabled
  }.to_json)

  separation_enabled
end
```

Organization Setting:

```
def sub_item_separation_enabled?(org_id)
  setting = Org::OrgSetting.find_by(
    org_id: org_id,
    section: 'seperate_pre_req_item',
    setting_type: 'seperate_pre_req_item'
  )

  return false unless setting.present?
  setting.value == 't'
end
```

Separation Process:

```
def separate_by_sub_items(pre_req_ids)
  separated_item_count = 0

  pre_req_ids.each do |pre_req_id|
    pre_req = Org::PreRequisition.find_by_id(pre_req_id)
    next unless pre_req.present?

    pre_req_items = Org::PreRequisitionItem.where(org_pre_requisition_id: pre_req_id)

    pre_req_items.each do |item|
      inventories = item.org_pre_requisition_inventories
      next if inventories.count <= MIN_INVENTORY_COUNT

      grouped_inventories = group_inventories_by_identifiers(inventories)
      next if grouped_inventories.length <= MIN_INVENTORY_COUNT

      # Perform separation logic...
    end
  end
end
```

Grouping Logic:

```
def group_inventories_by_identifiers(inventories)
  inventories.group_by do |inventory|
```

```
    [
      inventory.lot_number.to_s.upcase.strip,
      inventory.serial_number.to_s.upcase.strip,
      inventory.exp_date&.strftime(DATE_FORMAT) || ''
    ]
  end.values
end
```

6.1.4. X12 File Creation

6.1.4.1. Overview X12 file creation generates standardized EDI files for external system integration, following the EDI 850 (Purchase Order) format.

File Creation Process:

```
def create_files(pre_req_ids)
  file_paths = []
  pre_req_ids.each do |pre_req_id|
    file_paths << create_file_x12(@params['org_id'], pre_req_id)
  end

  @logger.info({
    action: 'files_created',
    file_paths: file_paths,
    count: file_paths.length
  }.to_json)

  file_paths.compact
end
```

X12 File Structure: - **Interchange Control**: ISA/IEA segments - **Functional Group**: GS/GE segments - **Transaction Set**: ST/SE segments - **Purchase Order**: BEG, DTM, N1 segments - **Line Items**: PO1, PID segments - **Notes**: NTE segments for inventories and comments - **Attachments**: ATC segments for file references

6.1.4.2. Workflow Data Preparation:

```
def create_file_x12(org_id, pre_requisition_id)
  data = build_data_for_x12(pre_requisition_id, @current_user)
  dir_path = ENV['INTEGRATION_PATH']
  file_name = build_x12_file_name(org_id, DateTime.now).to_s

  x12_process = X12ProcessImproved.new(data, dir_path, file_name)
  return if x12_process.empty?
```

```
  x12_process.export
end
```

File Generation Process: 1. **Data Assembly**: Collect all pre-requisition data 2. **File Naming**: Generate unique file name with timestamp 3. **X12 Processing**: Create X12ProcessImproved instance 4. **Content Generation**: Build X12 segments and content 5. **File Writing**: Write complete file to integration directory 6. **Permission Setting**: Set appropriate file permissions

X12 Segment Building:

```
def export
  content_buffer = []

  # Build header segments
  content_buffer << build_isa
  content_buffer << build_gs
  content_buffer << build_st
  content_buffer << build_beg
  content_buffer << build_dtm
  content_buffer << build_n1by
  content_buffer << build_n2
  content_buffer << build_n1se
  content_buffer.concat(build_n1dt_and_n1st)
  content_buffer << build_nte_header

  # Process items
  @items.each do |item|
    content_buffer << build_po1(item_hash)
    content_buffer << build_pid(item_hash)
    # Process inventories, comments, attachments...
  end

  # Build footer segments
  content_buffer << build_ctt(item_count)
  content_buffer << build_se
  content_buffer << build_ge
  content_buffer << build_iea

  # Write file
  File.write(full_name, content_buffer.join(@segment_delim))
end
```

Error Handling: - Validates data before file creation - Handles file system errors gracefully - Deletes partial files on failure - Logs all file creation activities

File Management: - Sets file permissions to 0o664 - Uses buffered writing for performance - Handles large files efficiently - Provides audit trail for file

operations

This detailed design document provides comprehensive coverage of the submit pre-requisition functionality, including all major components, processes, and implementation details necessary for system understanding and maintenance.