

ZIP Batch Upload Support - Architecture Design Plan

Executive Summary

This document outlines the architectural design for adding ZIP batch upload functionality to the document processing system. The solution extends the existing single PDF upload endpoint to accept ZIP files containing multiple PDF documents while maintaining backward compatibility and leveraging the existing asynchronous processing pipeline.

Current Architecture Analysis

Existing Upload Flow

1. **API Endpoint:** `POST /client/{id}/document` accepts single PDF files via multipart form
2. **Upload Handler:** `api/queryAPI/documents.go:UploadDocument()` - no content type validation
3. **Upload Service:** `internal/document/upload/get.go:Upload()` - stores file in S3 and creates DB record
4. **Processing Pipeline:** `S3 → SQS → storeEventRunner → docInitRunner → docSyncRunner → docCleanRunner → docTextRunner → queryRunner`

Current Constraints

- **OpenAPI Spec:** Defines `contentType: application/pdf` (line 433)
- **Content Type Validation:** Only occurs downstream in `docCleanRunner` via `internal/document/clean/contentType.go`
- **MIME Type Support:** System only processes PDF files
(`AcceptMimeTypes = []documenttypes.MimeType{documenttypes.MimeTypePDF}`)
- **Single File Processing:** Each upload creates one document record and follows individual processing flow

Design Decision: Extend Existing Endpoint

Recommendation: Extend the existing `POST /client/{id}/document` endpoint rather than creating a new one.

Rationale: - Maintains API simplicity and backward compatibility - Leverages existing authentication, validation, and error handling - Content type detection can differentiate between PDF and ZIP files - Follows principle of least surprise for API consumers

Architecture Design

1. API Layer Changes

OpenAPI Specification Updates

```
# serviceAPIs/queryAPI.yaml - line 432-433
encoding:
  file:
    contentType: "application/pdf,application/zip" # Updated to
               accept both
    style: form
    explode: false
```

Content Type Detection

- **PDF Files:** Process as currently implemented
- **ZIP Files:** Detect via MIME type application/zip or file extension
- **Invalid Files:** Reject with appropriate error message

2. Upload Handler Enhancement

File Type Router (api/queryAPI/documents.go)

```
func (s *Controllers) UploadDocument(ctx echo.Context, clientId
    ClientID) error {
    // Existing file extraction logic...

    // Detect file type
    fileType := detectFileType(file.Header.Get("Content-Type"),
        file.Filename)

    switch fileType {
    case FileTypePDF:
        return s.handleSinglePDFUpload(ctx, clientId, file)
    case FileTypeZIP:
        return s.handleZIPBatchUpload(ctx, clientId, file)
    default:
        return echo.NewHTTPError(http.StatusBadRequest,
            "unsupported file type")
    }
}
```

3. Batch Upload Processing

ZIP Processing Service (internal/document/batch/)

```
type BatchUploadService struct {
    cfg          ServiceConfig
    singleUpload *documentupload.Service
}

func (s *BatchUploadService) ProcessZIP(ctx context.Context,
    params BatchUploadParams) (*BatchUploadResult, error) {
    // 1. Extract ZIP contents to temporary storage
    // 2. Validate each PDF file
    // 3. Create batch upload record
    // 4. Process each PDF individually via existing upload
    //    service
    // 5. Return batch status with individual document IDs
}
```

Batch Upload Flow

1. **ZIP Extraction:** Extract ZIP contents to temporary S3 location
2. **PDF Validation:** Validate each extracted file is a valid PDF
3. **Batch Record Creation:** Create batch upload record in database
4. **Individual Processing:** Process each PDF through existing single upload flow
5. **Status Tracking:** Track overall batch progress and individual document status

4. Database Schema Changes

New Tables

```
-- Batch upload tracking
CREATE TABLE documentBatchUploads (
    id uuid PRIMARY KEY DEFAULT uuid_generate_v7(),
    clientId varchar(255) NOT NULL,
    originalFilename text NOT NULL,
    totalDocuments integer NOT NULL,
    processedDocuments integer DEFAULT 0,
    successfulDocuments integer DEFAULT 0,
    failedDocuments integer DEFAULT 0,
    status batchUploadStatus NOT NULL DEFAULT 'processing',
    createdAt timestamp NOT NULL DEFAULT NOW(),
    completedAt timestamp,
    FOREIGN KEY (clientId) REFERENCES clients(clientId)
);

CREATE TYPE batchUploadStatus AS ENUM (
    'processing',
    'completed',
    'failed',
    'cancelled'
```

```

);

-- Link individual documents to batch
CREATE TABLE documentBatchItems (
  id uuid PRIMARY KEY DEFAULT uuid_generate_v7(),
  batchId uuid NOT NULL,
  documentId uuid, -- NULL if processing failed
  originalFilename text NOT NULL,
  status batchItemStatus NOT NULL DEFAULT 'pending',
  errorMessage text,
  createdAt timestamp NOT NULL DEFAULT NOW(),
  FOREIGN KEY (batchId) REFERENCES documentBatchUploads(id),
  FOREIGN KEY (documentId) REFERENCES documents(id)
);

CREATE TYPE batchItemStatus AS ENUM (
  'pending',
  'processing',
  'completed',
  'failed'
);

```

Database Queries (internal/database/queries/batch.sql)

```

-- name: CreateBatchUpload :one
INSERT INTO documentBatchUploads (clientId, originalFilename,
  totalDocuments)
VALUES ($1, $2, $3) RETURNING id;

-- name: GetBatchUploadStatus :one
SELECT id, clientId, originalFilename, totalDocuments,
  processedDocuments,
  successfulDocuments, failedDocuments, status, createdAt,
  completedAt
FROM documentBatchUploads WHERE id = $1;

-- name: UpdateBatchProgress :exec
UPDATE documentBatchUploads
SET processedDocuments = $2, successfulDocuments = $3,
  failedDocuments = $4,
  status = $5, completedAt = $6
WHERE id = $1;

```

5. Asynchronous Processing Strategy

Processing Approach: Hybrid Async

- **Initial Response:** Synchronous validation and batch creation (~2-5 seconds)
- **Document Processing:** Asynchronous individual document processing
- **Status Monitoring:** Real-time batch progress tracking

Processing Flow

1. **Immediate Response** (Synchronous):
 - Validate ZIP file structure
 - Extract and count PDF files
 - Create batch record
 - Return batch ID and initial status
2. **Background Processing** (Asynchronous):
 - Process each PDF through existing pipeline
 - Update batch progress in real-time
 - Handle individual document failures gracefully

Response Format

```
{
  "batch_id": "019580df-ef65-7676-8de9-94435a93337a",
  "total_documents": 15,
  "status": "processing",
  "individual_documents": [
    {
      "filename": "document1.pdf",
      "document_id": "019580df-b3f8-7348-9ab1-1e55b3f18ed7",
      "status": "completed"
    },
    {
      "filename": "document2.pdf",
      "status": "processing"
    }
  ]
}
```

6. Status Monitoring System

New API Endpoints

```
# Get batch upload status
GET /batch/{batch_id}:
responses:
  200:
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/BatchUploadStatus'

# List batch uploads for client
GET /client/{id}/batch:
responses:
  200:
    content:
      application/json:
        schema:
          type: array
```

```
items:
  $ref: '#/components/schemas/BatchUploadSummary'
```

Real-time Updates

- **WebSocket Support:** Optional real-time progress updates
- **Polling Strategy:** Clients can poll batch status endpoint
- **Event Notifications:** Integration with existing SQS event system

7. Error Handling Strategy

Error Categories

1. **ZIP Level Errors:** Corrupt ZIP, no PDF files found, ZIP too large
2. **Individual PDF Errors:** Invalid PDF, processing failures, duplicate detection
3. **System Errors:** Storage failures, database errors, queue failures

Error Handling Approach

- **Partial Success:** Allow batch to succeed even if some PDFs fail
- **Error Isolation:** Individual PDF failures don't affect other documents in batch
- **Detailed Reporting:** Provide specific error messages for each failed document
- **Retry Logic:** Implement retry mechanism for transient failures

8. Implementation Plan

Phase 1: Core Infrastructure (Sprint 1-2)

1. **Database Schema:** Create batch upload tables and queries
2. **ZIP Processing Service:** Implement ZIP extraction and validation
3. **Batch Upload Service:** Create batch management service
4. **API Handler Updates:** Extend existing upload endpoint

Phase 2: Processing Pipeline (Sprint 3-4)

1. **Async Processing:** Implement background batch processing
2. **Status Tracking:** Real-time progress updates
3. **Error Handling:** Comprehensive error management
4. **Testing:** Unit and integration tests

Phase 3: Monitoring & API (Sprint 5)

1. **Status API Endpoints:** Batch status and listing endpoints
2. **OpenAPI Updates:** Update API specification
3. **Documentation:** Update API documentation
4. **Performance Testing:** Load and stress testing

Phase 4: Production Readiness (Sprint 6)

1. **Monitoring:** Metrics and logging integration

2. **Security Review:** Security assessment and validation
3. **Deployment:** Production deployment and validation
4. **Documentation:** User guides and operational runbooks

9. Risk Assessment & Mitigation

Technical Risks

1. **Memory Usage:** Large ZIP files could consume significant memory
 - **Mitigation:** Stream-based ZIP processing, configurable limits
2. **Storage Costs:** Temporary storage for ZIP extraction
 - **Mitigation:** Cleanup processes, storage lifecycle policies
3. **Processing Time:** Large batches could overwhelm system
 - **Mitigation:** Batch size limits, rate limiting, queue management

Operational Risks

1. **Partial Failures:** Complex error scenarios with mixed success/failure
 - **Mitigation:** Clear status reporting, retry mechanisms
2. **Resource Exhaustion:** High concurrent batch uploads
 - **Mitigation:** Resource monitoring, throttling, autoscaling

10. Success Metrics

Performance Metrics

- **Upload Success Rate:** >95% successful batch uploads
- **Processing Time:** <30 seconds for 10-document batches
- **System Throughput:** Support 100+ concurrent batch uploads
- **Error Rate:** <5% individual document failures

User Experience Metrics

- **API Response Time:** <5 seconds for batch upload initiation
- **Status Update Latency:** <10 seconds for progress updates
- **Documentation Completeness:** 100% API endpoint documentation

Conclusion

This design provides a comprehensive solution for ZIP batch upload support that:

- **Maintains Compatibility:** Existing single PDF upload functionality unchanged
- **Leverages Infrastructure:** Reuses existing processing pipeline and services
- **Provides Flexibility:** Supports both synchronous validation and asynchronous processing
- **Ensures Reliability:** Comprehensive error handling and status monitoring
- **Scales Effectively:** Asynchronous processing prevents system overload

The phased implementation approach allows for incremental delivery and validation, minimizing risk while providing immediate value to users requiring batch document upload capabilities.