Document Replication Agent — Source Code

Project G Beta — Generated 2026-02-15

DocumentReplicationAgent.tsx

src/components/crewos/documentReplication/DocumentReplicationAgent.tsx

```
import { useState, useCallback } from 'react';
import { ChevronRight, FileText, Layers, Type } from 'lucide-react';
import { DocumentUploader } from './DocumentUploader';
import { DocumentViewer } from './DocumentViewer';
import { FieldMapper } from './FieldMapper';
import { DataEntryForm } from './DataEntryForm';
import { DocumentGenerator } from './DocumentGenerator';
import { uploadDocument } from '../../../services/documentReplication/api';
import type { DetectedField } from '../../../services/documentReplication/types';
import './DocumentReplication.css';

type Step = 'upload' | 'map' | 'data' | 'generate';

interface DocumentReplicationAgentProps {
  userId: string;
  onBack?: () => void;
}

export function DocumentReplicationAgent({ userId, onBack }: DocumentReplicationAgentProps) {
  const [step, setStep] = useState<Step>('upload');
  const [file, setFile] = useState<File | null>(null);
  const [templateId, setTemplateId] = useState<string | null>(null);
  const [fields, setFields] = useState<DetectedField[]>([]);
  const [data, setData] = useState<Record<string, string>>({});

  const handleUploadComplete = useCallback(
    (uploadedFile: File, id: string, detectedFields: DetectedField[]) => {
      setFile(uploadedFile);
      setTemplateId(id);
      setFields(detectedFields.map((f) => ({ ...f, userConfirmed: f.confidence >= 0.7 })));
      setStep('map');
    },
    []
  );

  const goToDataEntry = useCallback(() => {
    const confirmed = fields.filter((f) => f.userConfirmed !== false);
    if (confirmed.length > 0) {
      setData((prev) => {
        const next = { ...prev };
        confirmed.forEach((f) => {
          if (f.sampleValue && next[f.id] === undefined) next[f.id] = f.sampleValue;
        });
        return next;
      });
      setStep('data');
    }
  }, [fields]);

  const goToGenerate = useCallback(() => setStep('generate'), []);

  return (
    <div className="replication-agent">
      <div className="replication-agent-header">
        {onBack && (
          <button type="button" className="replication-back" onClick={onBack}>
            <ChevronRight size={16} style={{ transform: 'rotate(180deg)' }} />
            Back
          </button>
        )}
        <h2 className="replication-agent-title">Document Replication</h2>
        <p className="replication-agent-subtitle">
          Upload a template, map variable fields, fill in data, and generate new documents.
```

```
          </p>
          <div className="replication-steps">
            {[
              { id: 'upload' as const, label: 'Upload', icon: FileText },
              { id: 'map' as const, label: 'Map fields', icon: Layers },
              { id: 'data' as const, label: 'Data', icon: Type },
              { id: 'generate' as const, label: 'Generate', icon: FileText },
            ].map((s, i) => (
              <div
                key={s.id}
                className={`replication-step ${step === s.id ? 'active' : ''} ${stepOrder(step) > i ? 'done' :
              >
                {i > 0 && <span className="replication-step-connector" />}
                <span className="replication-step-dot">
                  <s.icon size={14} />
                </span>
                <span>{s.label}</span>
              </div>
            ))}
          </div>
        </div>

        <div className="replication-agent-content">
          {step === 'upload' && (
            <DocumentUploader
              onUploadComplete={handleUploadComplete}
              uploadDocument={uploadDocument}
              userId={userId}
            />
          )}

          {step === 'map' && templateId && (
            <div className="replication-map-layout">
              <div className="replication-map-viewer">
                <DocumentViewer file={file} />
              </div>
              <div className="replication-map-sidebar">
                <FieldMapper fields={fields} onFieldsChange={setFields} />
                <button type="button" className="replication-btn-primary replication-next" onClick={goToDataEnt
                  Continue to data entry
                  <ChevronRight size={16} />
                </button>
              </div>
            </div>
          )}

          {step === 'data' && (
            <div className="replication-data-layout">
              <DataEntryForm
                fields={fields}
                data={data}
                onDataChange={setData}
              />
              <button type="button" className="replication-btn-primary replication-next" onClick={goToGenerate}
                Continue to generate
                <ChevronRight size={16} />
              </button>
            </div>
          )}

          {step === 'generate' && templateId && (
            <div className="replication-generate-layout">
              <DataEntryForm fields={fields} data={data} onDataChange={setData} disabled />
              <DocumentGenerator
                templateId={templateId}
                fields={fields}
                data={data}
```

```
              suggestedFileName={file ? file.name.replace(/\.[^.]+$/, '') + '_generated.pdf' : undefined}
            />
          </div>
        )}
      </div>
    </div>
  );
}

function stepOrder(step: Step): number {
  const order: Record<Step, number> = { upload: 0, map: 1, data: 2, generate: 3 };
  return order[step] ?? 0;
}
```

DocumentUploader.tsx

src/components/crewos/documentReplication/DocumentUploader.tsx

```
import { useCallback, useState } from 'react';
import { Upload, FileText, Loader2, AlertCircle } from 'lucide-react';

const ACCEPT = '.pdf,.docx,.doc,.txt,application/pdf,application/vnd.openxmlformats-officedocument.wordproces
const MAX_MB = 10;

interface DocumentUploaderProps {
  onUploadComplete: (file: File, templateId: string, fields: import('../../../services/documentReplication/ty
  uploadDocument: (file: File, userId: string) => Promise<{ templateId: string; fields?: import('../../../ser
  userId: string;
}

export function DocumentUploader({ onUploadComplete, uploadDocument, userId }: DocumentUploaderProps) {
  const [dragOver, setDragOver] = useState(false);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  const handleFile = useCallback(
    async (file: File) => {
      if (file.size > MAX_MB * 1024 * 1024) {
        setError(`File must be under ${MAX_MB}MB`);
        return;
      }
      setError(null);
      setLoading(true);
      try {
        const { templateId, fields } = await uploadDocument(file, userId);
        onUploadComplete(file, templateId, fields || []);
      } catch (e) {
        const msg = e instanceof Error ? e.message : 'Upload failed';
        const isNetwork = /fetch|network|connection|reset/i.test(msg) || msg === 'Failed to fetch';
        setError(isNetwork
          ? 'Cannot reach the server. Start the backend: in the server/ folder run npm run dev (port 3001).'
          : msg);
      } finally {
        setLoading(false);
      }
    },
    [onUploadComplete, uploadDocument, userId]
  );

  const onDrop = useCallback(
    (e: React.DragEvent) => {
      e.preventDefault();
      setDragOver(false);
      const file = e.dataTransfer.files[0];
      if (file) handleFile(file);
    },
    [handleFile]
  );

  const onInputChange = useCallback(
    (e: React.ChangeEvent<HTMLInputElement>) => {
      const file = e.target.files?.[0];
      if (file) handleFile(file);
      e.target.value = '';
    },
    [handleFile]
  );

  return (
    <div className="replication-uploader">
      <div
```

```
        className={`replication-dropzone ${dragOver ? 'drag-over' : ''} ${loading ? 'loading' : ''}`}
        onDragOver={(e) => { e.preventDefault(); setDragOver(true); }}
        onDragLeave={() => setDragOver(false)}
        onDrop={onDrop}
      >
        <input
          type="file"
          accept={ACCEPT}
          onChange={onInputChange}
          disabled={loading}
          className="replication-dropzone-input"
        />
        {loading ? (
          <>
            <Loader2 size={32} className="replication-spin" />
            <p>Analyzing document and detecting fields…</p>
          </>
        ) : (
          <>
            <Upload size={32} />
            <p>Drop a PDF or DOCX here, or click to browse</p>
            <span className="replication-dropzone-hint">Max {MAX_MB}MB. We'll extract text and suggest variab
          </>
        )}
      </div>
      {error && (
        <div className="replication-error">
          <AlertCircle size={16} />
          <span>{error}</span>
        </div>
      )}
    </div>
  );
}
```

FieldMapper.tsx

src/components/crewos/documentReplication/FieldMapper.tsx

```
import { useCallback } from 'react';
import { Check, X, Plus, Tag } from 'lucide-react';
import type { DetectedField, FieldType } from '../../../services/documentReplication/types';

const FIELD_TYPES: FieldType[] = ['text', 'number', 'date', 'currency', 'email', 'phone', 'id', 'percentage']

interface FieldMapperProps {
  fields: DetectedField[];
  onFieldsChange: (fields: DetectedField[]) => void;
  disabled?: boolean;
}

export function FieldMapper({ fields, onFieldsChange, disabled }: FieldMapperProps) {
  const confirm = useCallback(
    (id: string) => {
      onFieldsChange(
        fields.map((f) => (f.id === id ? { ...f, userConfirmed: true } : f))
      );
    },
    [fields, onFieldsChange]
  );

  const reject = useCallback(
    (id: string) => {
      onFieldsChange(fields.filter((f) => f.id !== id));
    },
    [fields, onFieldsChange]
  );

  const setType = useCallback(
    (id: string, type: FieldType) => {
      onFieldsChange(
        fields.map((f) => (f.id === id ? { ...f, type } : f))
      );
    },
    [fields, onFieldsChange]
  );

  const setName = useCallback(
    (id: string, name: string) => {
      onFieldsChange(
        fields.map((f) => (f.id === id ? { ...f, name: name.trim() || f.name } : f))
      );
    },
    [fields, onFieldsChange]
  );

  const addField = useCallback(() => {
    const newField: DetectedField = {
      id: `manual-${Date.now()}`,
      name: 'New field',
      type: 'text',
      page: 1,
      confidence: 1,
      aiSuggested: false,
      userConfirmed: true,
    };
    onFieldsChange([...fields, newField]);
  }, [fields, onFieldsChange]);

  return (
    <div className="replication-field-mapper">
      <div className="replication-field-mapper-header">
```

```
        <h4>Variable fields</h4>
        <button type="button" className="replication-btn-add" onClick={addField} disabled={disabled}>
          <Plus size={14} />
          Add field
        </button>
      </div>
      <ul className="replication-field-list">
        {fields.map((f) => (
          <li key={f.id} className={`replication-field-item ${f.userConfirmed ? 'confirmed' : ''}`}>
            <div className="replication-field-row">
              <input
                type="text"
                value={f.name}
                onChange={(e) => setName(f.id, e.target.value)}
                className="replication-field-name"
                placeholder="Field name"
                disabled={disabled}
              />
              <select
                value={f.type}
                onChange={(e) => setType(f.id, e.target.value as FieldType)}
                className="replication-field-type"
                disabled={disabled}
              >
                {FIELD_TYPES.map((t) => (
                  <option key={t} value={t}>
                    {t}
                  </option>
                ))}
              </select>
              {f.aiSuggested && (
                <span className="replication-field-confidence">
                  {Math.round(f.confidence * 100)}%
                </span>
              )}
              {!f.userConfirmed && (
                <>
                  <button type="button" onClick={() => confirm(f.id)} className="replication-field-btn accept
                    <Check size={14} />
                  </button>
                  <button type="button" onClick={() => reject(f.id)} className="replication-field-btn reject"
                    <X size={14} />
                  </button>
                </>
              )}
            </div>
            {f.sampleValue && (
              <div className="replication-field-sample">Sample: {f.sampleValue}</div>
            )}
          </li>
        ))}
      </ul>
      {fields.length === 0 && (
        <p className="replication-field-empty">No fields yet. Add one manually or re-upload a document.</p>
      )}
    </div>
  );
}
```

DataEntryForm.tsx

src/components/crewos/documentReplication/DataEntryForm.tsx

```
import { useCallback } from 'react';
import type { DetectedField, FieldType } from '../../../services/documentReplication/types';

interface DataEntryFormProps {
  fields: DetectedField[];
  data: Record<string, string>;
  onDataChange: (data: Record<string, string>) => void;
  disabled?: boolean;
}

export function DataEntryForm({ fields, data, onDataChange, disabled }: DataEntryFormProps) {
  const confirmed = fields.filter((f) => f.userConfirmed !== false);

  const setValue = useCallback(
    (fieldId: string, value: string) => {
      onDataChange({ ...data, [fieldId]: value });
    },
    [data, onDataChange]
  );

  const inputType = (type: FieldType): string => {
    if (type === 'date') return 'date';
    if (type === 'number' || type === 'currency' || type === 'percentage') return 'text';
    if (type === 'email') return 'email';
    return 'text';
  };

  return (
    <div className="replication-data-form">
      <h4>Enter values</h4>
      <div className="replication-data-fields">
        {confirmed.map((f) => (
          <div key={f.id} className="replication-data-field">
            <label htmlFor={f.id}>{f.name}</label>
            {typeNeedsTextarea(f.type) ? (
              <textarea
                id={f.id}
                value={data[f.id] ?? ''}
                onChange={(e) => setValue(f.id, e.target.value)}
                placeholder={f.sampleValue}
                rows={2}
                disabled={disabled}
                className="replication-input replication-textarea"
              />
            ) : (
              <input
                id={f.id}
                type={inputType(f.type)}
                value={data[f.id] ?? ''}
                onChange={(e) => setValue(f.id, e.target.value)}
                placeholder={f.sampleValue}
                disabled={disabled}
                className="replication-input"
              />
            )}
          </div>
        ))}
      </div>
    </div>
  );
}

function typeNeedsTextarea(type: FieldType): boolean {
```

```
  return type === 'text';
}
```

DocumentGenerator.tsx

src/components/crewos/documentReplication/DocumentGenerator.tsx

```
import { useState, useCallback } from 'react';
import { FileDown, Loader2, Eye, AlertCircle } from 'lucide-react';
import type { DetectedField, OutputFormat } from '../../../services/documentReplication/types';
import { generateReplicationDocument } from '../../../services/documentReplication/api';

interface DocumentGeneratorProps {
  templateId: string;
  fields: DetectedField[];
  data: Record<string, string>;
  suggestedFileName?: string;
  disabled?: boolean;
}

export function DocumentGenerator({
  templateId,
  fields,
  data,
  suggestedFileName,
  disabled,
}: DocumentGeneratorProps) {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);
  const [previewUrl, setPreviewUrl] = useState<string | null>(null);

  const confirmedFields = fields.filter((f) => f.userConfirmed !== false);

  const handleGenerate = useCallback(
    async (format: OutputFormat = 'pdf') => {
      if (!templateId || confirmedFields.length === 0) return;
      setError(null);
      setLoading(true);
      setPreviewUrl(null);
      try {
        const res = await generateReplicationDocument({
          templateId,
          fieldMappings: confirmedFields,
          data,
          outputFormat: format,
          fileName: suggestedFileName || `Generated_${Date.now()}.pdf`,
        });
        if (res.generatedUrl) {
          setPreviewUrl(res.generatedUrl);
        } else {
          setError(res.error || 'No download URL returned');
        }
      } catch (e) {
        setError(e instanceof Error ? e.message : 'Generation failed');
      } finally {
        setLoading(false);
      }
    },
    [templateId, confirmedFields, data, suggestedFileName]
  );

  const handleDownload = useCallback(() => {
    if (!previewUrl) return;
    const a = document.createElement('a');
    a.href = previewUrl;
    a.download = suggestedFileName || `document_${Date.now()}.pdf`;
    a.click();
  }, [previewUrl, suggestedFileName]);

  return (
```

```
    <div className="replication-generator">
      <h4>Generate document</h4>
      <div className="replication-generator-actions">
        <button
          type="button"
          className="replication-btn-primary"
          onClick={() => handleGenerate('pdf')}
          disabled={disabled || loading || confirmedFields.length === 0}
        >
          {loading ? (
            <>
              <Loader2 size={16} className="replication-spin" />
              Generating…
            </>
          ) : (
            <>
              <FileDown size={16} />
              Generate PDF
            </>
          )}
        </button>
      </div>
      {error && (
        <div className="replication-error">
          <AlertCircle size={16} />
          <span>{error}</span>
        </div>
      )}
      {previewUrl && (
        <div className="replication-preview">
          <p>Preview ready.</p>
          <div className="replication-preview-actions">
            <a href={previewUrl} target="_blank" rel="noopener noreferrer" className="replication-btn-seconda
              <Eye size={14} />
              Open in new tab
            </a>
            <button type="button" className="replication-btn-primary" onClick={handleDownload}>
              <FileDown size={14} />
              Download PDF
            </button>
          </div>
          <iframe
            src={previewUrl}
            title="Generated document preview"
            className="replication-preview-iframe"
          />
        </div>
      )}
    </div>
  );
}
```

DocumentViewer.tsx

src/components/crewos/documentReplication/DocumentViewer.tsx

```
import { useState, useMemo } from 'react';
import { ZoomIn, ZoomOut, Maximize2 } from 'lucide-react';

interface DocumentViewerProps {
  file: File | null;
  fileUrl?: string | null;
  className?: string;
}

export function DocumentViewer({ file, fileUrl, className = '' }: DocumentViewerProps) {
  const [zoom, setZoom] = useState(1);
  const objectUrl = useMemo(() => {
    if (fileUrl && (fileUrl.startsWith('blob:') || fileUrl.startsWith('data:'))) return fileUrl;
    if (file) return URL.createObjectURL(file);
    return null;
  }, [file, fileUrl]);

  if (!objectUrl && !file) {
    return (
      <div className={`replication-viewer replication-viewer-empty ${className}`}>
        <p>No document to display</p>
      </div>
    );
  }

  const isPdf = file?.type === 'application/pdf' || (typeof fileUrl === 'string' && fileUrl.includes('pdf'));
  const isImage = file?.type?.startsWith('image/');

  return (
    <div className={`replication-viewer ${className}`}>
      <div className="replication-viewer-toolbar">
        <button type="button" onClick={() => setZoom((z) => Math.max(0.5, z - 0.25))} aria-label="Zoom out">
          <ZoomOut size={18} />
        </button>
        <span>{Math.round(zoom * 100)}%</span>
        <button type="button" onClick={() => setZoom((z) => Math.min(2, z + 0.25))} aria-label="Zoom in">
          <ZoomIn size={18} />
        </button>
      </div>
      <div className="replication-viewer-content" style={{ transform: `scale(${zoom})` }}>
        {isPdf && (
          <object
            data={objectUrl || undefined}
            type="application/pdf"
            className="replication-viewer-object"
            title="PDF preview"
          >
            <p>PDF preview not supported. Download the file to view.</p>
          </object>
        )}
        {isImage && (
          <img src={objectUrl || undefined} alt="Document" className="replication-viewer-img" />
        )}
        {!isPdf && !isImage && objectUrl && (
          <iframe src={objectUrl} title="Document" className="replication-viewer-object" />
        )}
      </div>
    </div>
  );
}
```

index.ts (components)

src/components/crewos/documentReplication/index.ts

```
export { DocumentReplicationAgent } from './DocumentReplicationAgent';
export { DocumentUploader } from './DocumentUploader';
export { DocumentViewer } from './DocumentViewer';
export { FieldMapper } from './FieldMapper';
export { DataEntryForm } from './DataEntryForm';
export { DocumentGenerator } from './DocumentGenerator';
```

api.ts

src/services/documentReplication/api.ts

```
/**
 * Document Replication API client.
 * Calls backend /api/documents/* for analyze and generate.
 */

import type {
  AnalyzeDocumentRequest,
  AnalyzeDocumentResponse,
  GenerateDocumentRequest,
  GenerateDocumentResponse,
  DocumentTemplate,
} from './types';

const API_BASE = import.meta.env.VITE_API_URL || '/api';

async function request<T>(
  path: string,
  options: { method?: string; headers?: HeadersInit; body?: unknown } = {}
): Promise<T> {
  const { body, method, headers } = options;
  const fetchOptions: RequestInit = {
    method: method ?? 'GET',
    headers: {
      'Content-Type': 'application/json',
      ...(headers as Record<string, string>),
    },
  };
  if (body !== undefined) {
    fetchOptions.body = JSON.stringify(body);
  }
  const res = await fetch(`${API_BASE}/documents${path}`, fetchOptions);
  let data: T & { success?: boolean; error?: string };
  try {
    data = (await res.json()) as T & { success?: boolean; error?: string };
  } catch {
    throw new Error(res.status === 500 ? 'Server error (check backend terminal for details)' : res.statusText
  }
  if (!res.ok) {
    throw new Error((data as { error?: string }).error || res.statusText);
  }
  return data as T;
}

/** Response from POST /documents/upload */
interface UploadResponse {
  success: boolean;
  templateId: string;
  template?: DocumentTemplate;
  fields?: import('./types').DetectedField[];
  structure?: DocumentTemplate['structure'];
}

/**
 * Upload document (base64), run analysis, and get templateId + fields.
 */
export async function uploadDocument(
  file: File,
  userId: string
): Promise<{ templateId: string; template?: DocumentTemplate; fields?: import('./types').DetectedField[] }> {
  const base64 = await fileToBase64(file);
  const res = await request<UploadResponse>('/upload', {
    method: 'POST',
    body: {
```

```
      fileBase64: base64,
      fileName: file.name,
      mimeType: file.type,
      userId,
    },
  });
  if (!res.templateId) throw new Error('Upload did not return templateId');
  return {
    templateId: res.templateId,
    template: res.template,
    fields: res.fields,
  };
}

/**
 * Analyze document: extract text and run AI variable detection.
 * Can be called with storage URL (after upload) or with base64.
 */
export async function analyzeDocument(
  req: AnalyzeDocumentRequest
): Promise<AnalyzeDocumentResponse> {
  return request<AnalyzeDocumentResponse>('/analyze', {
    method: 'POST',
    body: req,
  });
}

/**
 * Generate a new document from template + field mappings + data.
 */
export async function generateReplicationDocument(
  req: GenerateDocumentRequest
): Promise<GenerateDocumentResponse> {
  return request<GenerateDocumentResponse>('/generate', {
    method: 'POST',
    body: req,
  });
}

function fileToBase64(file: File): Promise<string> {
  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = () => {
      const result = reader.result as string;
      const base64 = result.includes(',') ? result.split(',')[1] : result;
      resolve(base64 || '');
    };
    reader.onerror = reject;
    reader.readAsDataURL(file);
  });
}
```

types.ts

src/services/documentReplication/types.ts

```
/**
 * Document Replication Agent — types for upload, analysis, mapping, and generation.
 */

export type DocumentFormat = 'pdf' | 'docx' | 'txt' | 'image';

export type FieldType =
  | 'text'
  | 'number'
  | 'date'
  | 'currency'
  | 'email'
  | 'phone'
  | 'id'
  | 'percentage';

export interface BoundingBox {
  x: number;   // 0–1 or px
  y: number;
  width: number;
  height: number;
  page?: number;
}

export interface DetectedField {
  id: string;
  name: string;
  type: FieldType;
  boundingBox?: BoundingBox;
  page: number;
  confidence: number;
  aiSuggested: boolean;
  userConfirmed: boolean;
  sampleValue?: string;
}

export interface DocumentStructure {
  pages: number;
  extractedText?: string;
  layout?: unknown;
  tables?: unknown[];
  images?: unknown[];
}

export interface DocumentTemplate {
  id: string;
  userId: string;
  originalFileName: string;
  originalFormat: DocumentFormat;
  storageUrl: string;
  createdAt: number;
  fields: DetectedField[];
  structure: DocumentStructure;
}

export type OutputFormat = 'pdf' | 'docx' | 'txt' | 'html';

export interface GeneratedReplicationDocument {
  id: string;
  templateId: string;
  userId: string;
  outputFormat: OutputFormat;
  data: Record<string, string>;
```

```
  generatedUrl: string;
  createdAt: number;
  fileName: string;
}

export interface AnalyzeDocumentRequest {
  fileUrl?: string;
  fileBase64?: string;
  fileName: string;
  mimeType: string;
}

export interface AnalyzeDocumentResponse {
  success: boolean;
  templateId?: string;
  fields?: DetectedField[];
  structure?: DocumentStructure;
  error?: string;
}

export interface GenerateDocumentRequest {
  templateId: string;
  fieldMappings: DetectedField[];
  data: Record<string, string>;
  outputFormat: OutputFormat;
  fileName?: string;
}

export interface GenerateDocumentResponse {
  success: boolean;
  generatedUrl?: string;
  fileName?: string;
  error?: string;
}
```

index.ts (services)

src/services/documentReplication/index.ts

```
export * from './types';
export * from './api';
```

documents.ts (backend routes)

server/src/routes/documents.ts

```
/* ═══════════════════════════════════════════════════════
   Document Replication Agent — Upload, Analyze, Generate

   POST /api/documents/upload   — Upload file (base64), analyze, return templateId + fields
   POST /api/documents/analyze — Analyze only (file URL or base64)
   POST /api/documents/generate — Generate PDF from templateId + field mappings + data
   ═══════════════════════════════════════════════════════ */

import { Router, Request, Response } from 'express';
import {
  extractText,
  detectVariablesWithOpenAI,
  type DetectedField,
} from '../services/document-analyzer.js';
import {
  generatePdfFromFields,
  replicatePdfFromOriginal,
  type FieldMapping,
} from '../services/document-generator.js';

const router = Router();

/** In-memory store for uploaded templates (MVP). Key: templateId, value: { buffer, fields, fileName } */
const templateStore = new Map<
  string,
  { buffer: Buffer; fields: DetectedField[]; fileName: string; createdAt: number }
>();
const TTL_MS = 10 * 60 * 1000; // 10 minutes

function pruneExpired() {
  const now = Date.now();
  for (const [id, v] of templateStore.entries()) {
    if (now - v.createdAt > TTL_MS) templateStore.delete(id);
  }
}

/* ─── POST /upload ──────────────────────────────────── */
/**
 * Upload document: accept base64 file, extract text, run AI detection, store in memory, return templateId +
 */
router.post('/upload', async (req: Request, res: Response) => {
  try {
    pruneExpired();
    if (!req.body || typeof req.body !== 'object') {
      res.status(400).json({ success: false, error: 'Request body must be JSON with fileBase64 and fileName'
      return;
    }
    const { fileBase64, fileName, mimeType, userId } = req.body as {
      fileBase64?: string;
      fileName?: string;
      mimeType?: string;
      userId?: string;
    };
    if (!fileBase64 || !fileName) {
      res.status(400).json({ success: false, error: 'fileBase64 and fileName are required' });
      return;
    }

    const buffer = Buffer.from(fileBase64, 'base64');
    if (buffer.length > 10 * 1024 * 1024) {
      res.status(400).json({ success: false, error: 'File too large (max 10MB)' });
      return;
    }
```

```
    if (buffer.length === 0) {
      res.status(400).json({ success: false, error: 'Invalid file: decoded size is 0' });
      return;
    }

    let text: string;
    let pages: number;
    try {
      const extracted = await extractText(buffer, mimeType || 'application/octet-stream');
      text = extracted.text ?? '';
      pages = extracted.pages ?? 1;
    } catch (extractErr) {
      const msg = extractErr instanceof Error ? extractErr.message : 'Text extraction failed';
      console.error('Upload extractText error:', extractErr);
      res.status(500).json({ success: false, error: `Extraction failed: ${msg}` });
      return;
    }

    const structure = { pages, extractedText: text.slice(0, 5000) };

    let fields: DetectedField[];
    try {
      fields = await detectVariablesWithOpenAI(text, fileName);
    } catch (aiErr) {
      console.warn('AI detection failed, using fallback fields:', aiErr);
      fields = [];
    }
    if (fields.length === 0 && text.trim()) {
      fields = [{
        id: `field-0-${Date.now()}`,
        name: 'Document Content',
        type: 'text',
        page: 1,
        confidence: 0.5,
        aiSuggested: true,
        userConfirmed: false,
        sampleValue: text.slice(0, 200).trim() || undefined,
      }];
    }

    const templateId = `tpl-${Date.now()}-${Math.random().toString(36).slice(2, 9)}`;
    templateStore.set(templateId, {
      buffer,
      fields,
      fileName,
      createdAt: Date.now(),
    });

    res.json({
      success: true,
      templateId,
      template: {
        id: templateId,
        userId: userId || 'anonymous',
        originalFileName: fileName,
        originalFormat: mimeType?.includes('pdf') ? 'pdf' : mimeType?.includes('word') ? 'docx' : 'txt',
        storageUrl: '',
        createdAt: Date.now(),
        fields,
        structure,
      },
      fields,
      structure,
    });
  } catch (err) {
    const message = err instanceof Error ? err.message : 'Upload failed';
    const stack = err instanceof Error ? err.stack : undefined;
```

```
    console.error('Documents upload error:', message, stack || '');
    res.status(500).json({
      success: false,
      error: message,
    });
  }
});

/* ─── POST /analyze ──────────────────────────────────────── */
/**
 * Analyze only: same as upload but does not store. Returns fields + structure.
 */
router.post('/analyze', async (req: Request, res: Response) => {
  try {
    const { fileBase64, fileName, mimeType } = req.body as {
      fileBase64?: string;
      fileName?: string;
      mimeType?: string;
    };
    if (!fileBase64 || !fileName) {
      res.status(400).json({ success: false, error: 'fileBase64 and fileName are required' });
      return;
    }

    const buffer = Buffer.from(fileBase64, 'base64');
    const { text, pages } = await extractText(buffer, mimeType || 'application/octet-stream');
    const structure = { pages, extractedText: text.slice(0, 5000) };
    const fields = await detectVariablesWithOpenAI(text, fileName);

    res.json({
      success: true,
      fields,
      structure,
    });
  } catch (err) {
    console.error('Documents analyze error:', err);
    res.status(500).json({
      success: false,
      error: err instanceof Error ? err.message : 'Analysis failed',
    });
  }
});

/* ─── POST /generate ─────────────────────────────────────── */
/**
 * Generate PDF from templateId + field mappings + data.
 */
router.post('/generate', async (req: Request, res: Response) => {
  try {
    pruneExpired();
    const { templateId, fieldMappings, data, outputFormat, fileName } = req.body as {
      templateId?: string;
      fieldMappings?: Array<{ id: string; name: string; type?: string; page?: number }>;
      data?: Record<string, string>;
      outputFormat?: string;
      fileName?: string;
    };

    if (!templateId || !fieldMappings?.length) {
      res.status(400).json({
        success: false,
        error: 'templateId and fieldMappings are required',
      });
      return;
    }

    const stored = templateStore.get(templateId);
```

```
    const mappings: FieldMapping[] = fieldMappings.map((f) => {
      const storedField = stored?.fields?.find((sf) => sf.id === f.id);
      return {
        id: f.id,
        name: f.name,
        type: f.type || 'text',
        page: f.page,
        sampleValue: storedField?.sampleValue ?? (f as { sampleValue?: string }).sampleValue,
      };
    });
    const dataRecord = typeof data === 'object' && data !== null ? data : {};
    const outFileName = fileName || `Generated_${templateId}_${Date.now()}.pdf`;
    const format = (outputFormat || 'pdf').toLowerCase();

    if (format !== 'pdf') {
      res.status(400).json({
        success: false,
        error: 'Only PDF output is supported in MVP',
      });
      return;
    }

    const useReplicate =
      stored?.buffer &&
      stored.buffer.length > 0 &&
      mappings.some((m) => m.sampleValue && (dataRecord[m.id] ?? dataRecord[m.name] ?? '').trim() !== '');

    const pdfBuffer = useReplicate
      ? await replicatePdfFromOriginal(
          stored.buffer,
          mappings,
          dataRecord,
          stored.fileName || outFileName
        )
      : await generatePdfFromFields(
          mappings,
          dataRecord,
          stored?.fileName || outFileName
        );
    const base64 = pdfBuffer.toString('base64');
    const dataUrl = `data:application/pdf;base64,${base64}`;

    res.json({
      success: true,
      generatedUrl: dataUrl,
      fileName: outFileName.replace(/\.pdf$/i, '') + '.pdf',
    });
  } catch (err) {
    console.error('Documents generate error:', err);
    res.status(500).json({
      success: false,
      error: err instanceof Error ? err.message : 'Generation failed',
    });
  }
});

export { router as documentsRouter };
```

document-analyzer.ts (backend)

server/src/services/document-analyzer.ts

```
/**
 * Document Analyzer — Extract text from PDF/DOCX and detect variables via OpenAI.
 */

import { createRequire } from 'node:module';
import OpenAI from 'openai';
import { config } from '../config.js';

const require = createRequire(import.meta.url);

export interface DetectedField {
  id: string;
  name: string;
  type: string;
  boundingBox?: { x: number; y: number; width: number; height: number; page?: number };
  page: number;
  confidence: number;
  aiSuggested: boolean;
  userConfirmed: boolean;
  sampleValue?: string;
}

export interface DocumentStructure {
  pages: number;
  extractedText?: string;
}

let openai: OpenAI | null = null;
function getOpenAI(): OpenAI {
  if (!openai) openai = new OpenAI({ apiKey: config.openai.apiKey });
  return openai;
}

/**
 * Extract text from a PDF buffer using pdf-parse (CJS).
 */
export async function extractTextFromPdf(buffer: Buffer): Promise<{ text: string; pages: number }> {
  try {
    const pdfParseModule = require('pdf-parse');
    const pdfParse = typeof pdfParseModule === 'function' ? pdfParseModule : pdfParseModule?.default ?? pdfPa
    if (typeof pdfParse !== 'function') {
      console.warn('pdf-parse: no function export');
      return { text: '', pages: 1 };
    }
    const data = await pdfParse(buffer);
    const text = data?.text != null ? String(data.text) : '';
    const pages = typeof data?.numpages === 'number' ? data.numpages : 1;
    return { text, pages };
  } catch (e) {
    console.warn('pdf-parse failed, returning empty text:', e);
    return { text: '', pages: 1 };
  }
}

/**
 * Extract text from a DOCX buffer using mammoth.
 */
export async function extractTextFromDocx(buffer: Buffer): Promise<{ text: string; pages: number }> {
  try {
    const mammoth = await import('mammoth');
    const result = await mammoth.extractRawText({ buffer });
    const text = result.value || '';
    const pages = Math.max(1, Math.ceil(text.length / 3000));
```

```
    return { text, pages };
  } catch (e) {
    console.warn('mammoth failed, returning empty text:', e);
    return { text: '', pages: 1 };
  }
}

/**
 * Extract text from buffer based on mime type.
 */
export async function extractText(
  buffer: Buffer,
  mimeType: string
): Promise<{ text: string; pages: number }> {
  if (mimeType === 'application/pdf') return extractTextFromPdf(buffer);
  if (
    mimeType === 'application/vnd.openxmlformats-officedocument.wordprocessingml.document' ||
    mimeType === 'application/msword'
  ) {
    return extractTextFromDocx(buffer);
  }
  if (mimeType === 'text/plain') {
    const text = buffer.toString('utf-8');
    return { text, pages: Math.max(1, Math.ceil(text.length / 3000)) };
  }
  return { text: '', pages: 1 };
}

/**
 * Call OpenAI to detect variable-like fields in the extracted text.
 * Returns suggested fields with name, type, confidence, and sample value.
 */
export async function detectVariablesWithOpenAI(
  extractedText: string,
  fileName: string
): Promise<DetectedField[]> {
  const apiKey = config.openai?.apiKey;
  if (!apiKey || !extractedText.trim()) {
    return [];
  }

  const truncated = extractedText.slice(0, 12000);
  const prompt = `You are a document analysis assistant. Below is text extracted from a document named "${fil

Identify all likely VARIABLE or PLACEHOLDER fields — values that would change when creating another document

For each detected field return:
- name: short label (e.g. "Invoice Number", "Customer Name", "Date")
- type: one of text, number, date, currency, email, phone, id, percentage
- confidence: 0-1
- sampleValue: the value that appears in this document (or "..." if generic)

Return a JSON array of objects with keys: name, type, confidence, sampleValue. Use only the types listed. No

  try {
    const client = getOpenAI();
    const completion = await client.chat.completions.create({
      model: 'gpt-4o-mini',
      messages: [
        {
          role: 'user',
          content: `${prompt}\n\n---\nDocument text:\n${truncated}`,
        },
      ],
      response_format: { type: 'json_object' },
      max_tokens: 2000,
    });
```

```
    const content = completion.choices[0]?.message?.content?.trim();
    if (!content) return [];

    const parsed = JSON.parse(content) as { fields?: Array<{ name: string; type: string; confidence?: number;
    const list = Array.isArray(parsed.fields) ? parsed.fields : Array.isArray(parsed) ? parsed : [];
    const types = ['text', 'number', 'date', 'currency', 'email', 'phone', 'id', 'percentage'];

    return list.slice(0, 50).map((item, i) => ({
      id: `field-${i}-${Date.now()}`,
      name: String(item.name || `Field ${i + 1}`).slice(0, 80),
      type: types.includes(String(item.type)) ? item.type : 'text',
      page: 1,
      confidence: Math.min(1, Math.max(0, Number(item.confidence) ?? 0.8)),
      aiSuggested: true,
      userConfirmed: false,
      sampleValue: item.sampleValue != null ? String(item.sampleValue).slice(0, 200) : undefined,
    }));
  } catch (e) {
    console.error('OpenAI variable detection failed:', e);
    return [];
  }
}
```

document-generator.ts (backend)

server/src/services/document-generator.ts

```
/**
 * Document Generator — Produce PDF (and later DOCX) from template fields + data.
 * Can replicate the original PDF with only variable text replaced (overlay).
 */

import { createRequire } from 'node:module';
import { PDFDocument, rgb, StandardFonts } from 'pdf-lib';

const require = createRequire(import.meta.url);

export interface FieldMapping {
  id: string;
  name: string;
  type: string;
  page?: number;
  sampleValue?: string;
}

interface TextPosition {
  pageIndex: number; // 0-based
  x: number;
  y: number;
  width: number;
  height: number;
}

/**
 * Replicate the original PDF: keep full layout and replace only the variable
 * regions (found by sampleValue) with the new values from data.
 */
export async function replicatePdfFromOriginal(
  originalPdfBuffer: Buffer,
  fields: FieldMapping[],
  data: Record<string, string>,
  _fileName: string
): Promise<Buffer> {
  const doc = await PDFDocument.load(originalPdfBuffer, { ignoreEncryption: true });
  const pages = doc.getPages();
  const font = await doc.embedFont(StandardFonts.Helvetica);
  const fontSize = 11;

  // Get text positions from PDF.js for each field that has sampleValue
  const positionsByFieldId = await getTextPositionsFromPdf(originalPdfBuffer, fields);
  if (positionsByFieldId.size === 0) {
    // Fallback: return original unchanged if we couldn't find any text to replace
    const bytes = await doc.save();
    return Buffer.from(bytes);
  }

  for (const field of fields) {
    const newValue = (data[field.id] ?? data[field.name] ?? '').trim();
    const pos = positionsByFieldId.get(field.id);
    if (!pos || newValue === '') continue;

    const page = pages[pos.pageIndex];
    if (!page) continue;

    const { width: pageWidth, height: pageHeight } = page.getSize();
    // PDF y is from bottom; ensure we don't draw outside page
    const y = Math.max(0, Math.min(pos.y, pageHeight - 20));
    const pad = 2;
    const rectWidth = Math.min(pos.width + pad * 2, pageWidth - pos.x);
    const rectHeight = Math.min(pos.height + pad * 2, 24);
```

```
    // White overlay over old text
    page.drawRectangle({
      x: pos.x - pad,
      y: y - pad,
      width: rectWidth,
      height: rectHeight,
      color: rgb(1, 1, 1),
      opacity: 1,
    });
    // New text (clip to one line for simplicity)
    const line = newValue.replace(/\s+/g, ' ').slice(0, 80);
    page.drawText(line, {
      x: pos.x,
      y,
      size: fontSize,
      font,
      color: rgb(0.1, 0.1, 0.1),
    });
  }

  const bytes = await doc.save();
  return Buffer.from(bytes);
}

async function getTextPositionsFromPdf(
  buffer: Buffer,
  fields: FieldMapping[]
): Promise<Map<string, TextPosition>> {
  const result = new Map<string, TextPosition>();
  try {
    const pdfjsLib = require('pdfjs-dist/legacy/build/pdf.js');
    if (typeof pdfjsLib.GlobalWorkerOptions !== 'undefined') {
      pdfjsLib.GlobalWorkerOptions.workerSrc = '';
    }
    const loadingTask = pdfjsLib.getDocument({ data: buffer });
    const pdfDoc = await loadingTask.promise;
    const numPages = pdfDoc.numPages;

    for (const field of fields) {
      const searchText = normalizeForSearch(field.sampleValue ?? '');
      if (!searchText) continue;

      for (let pageIndex = 0; pageIndex < numPages; pageIndex++) {
        const page = await pdfDoc.getPage(pageIndex + 1);
        const textContent = await page.getTextContent();
        const items = textContent.items as Array<{ str: string; transform: number[]; width?: number; height?:

        const parts: string[] = [];
        for (const it of items) parts.push((it.str ?? '').trim());
        const fullText = parts.join(' ');
        const normalizedFull = normalizeForSearch(fullText);
        const idx = normalizedFull.indexOf(searchText);
        if (idx === -1) continue;

        // Map character range [idx, idx+searchText.length] back to item indices
        let charCount = 0;
        let startItemIdx = 0;
        let endItemIdx = 0;
        for (let i = 0; i < parts.length; i++) {
          const partNorm = normalizeForSearch(parts[i]);
          const nextCount = charCount + (partNorm.length + (i > 0 ? 1 : 0));
          if (charCount <= idx && idx < nextCount) startItemIdx = i;
          if (charCount < idx + searchText.length && idx + searchText.length <= nextCount) {
            endItemIdx = i;
            break;
          }
```

```
          charCount = nextCount;
          endItemIdx = i;
        }

        const itemsToUse = items.slice(startItemIdx, endItemIdx + 1);
        let minX = Infinity, minY = Infinity, maxX = -Infinity, maxY = -Infinity;
        for (const it of itemsToUse) {
          const t = it.transform || [];
          const x = t[4] ?? 0;
          const y = t[5] ?? 0;
          const w = (it.width ?? 0) * (t[0] ?? 1);
          const h = (it.height ?? 12) * (t[3] ?? 1);
          minX = Math.min(minX, x);
          minY = Math.min(minY, y);
          maxX = Math.max(maxX, x + w);
          maxY = Math.max(maxY, y + h);
        }
        if (minX !== Infinity) {
          result.set(field.id, {
            pageIndex,
            x: minX,
            y: minY,
            width: maxX - minX,
            height: maxY - minY,
          });
          break; // one position per field (first page where found)
        }
      }
    }
  } catch (e) {
    console.warn('getTextPositionsFromPdf failed, falling back to simple PDF:', e);
  }
  return result;
}

function normalizeForSearch(s: string): string {
  return s.replace(/\s+/g, ' ').trim().toLowerCase();
}

/**
 * Generate a simple PDF document listing all field names and their values.
 * Phase 2 can overlay onto the original template PDF for exact layout.
 */
export async function generatePdfFromFields(
  fields: FieldMapping[],
  data: Record<string, string>,
  fileName: string
): Promise<Buffer> {
  const doc = await PDFDocument.create();
  const font = await doc.embedFont(StandardFonts.Helvetica);
  const bold = await doc.embedFont(StandardFonts.HelveticaBold);
  const pageWidth = 612;
  const pageHeight = 792;
  const margin = 50;
  const lineHeight = 18;
  const titleSize = 16;
  const bodySize = 11;

  let page = doc.addPage([pageWidth, pageHeight]);
  let y = pageHeight - margin - 40;

  const drawText = (text: string, opts: { size?: number; bold?: boolean; indent?: number } = {}) => {
    const size = opts.size ?? bodySize;
    const fontToUse = opts.bold ? bold : font;
    const indent = opts.indent ?? 0;
    const lines = wrapText(text, fontToUse, size, pageWidth - margin * 2 - indent);
    for (const line of lines) {
```

```
      if (y < margin + lineHeight) {
        page = doc.addPage([pageWidth, pageHeight]);
        y = pageHeight - margin - 20;
      }
      page.drawText(line, {
        x: margin + indent,
        y,
        size,
        font: fontToUse,
        color: rgb(0.2, 0.2, 0.2),
      });
      y -= lineHeight;
    }
  };

  page.drawText(fileName || 'Generated Document', {
    x: margin,
    y: pageHeight - margin - 24,
    size: titleSize,
    font: bold,
    color: rgb(0.15, 0.15, 0.15),
  });
  y = pageHeight - margin - 56;

  page.drawText(`Generated: ${new Date().toISOString().slice(0, 10)}`, {
    x: margin,
    y,
    size: 10,
    font,
    color: rgb(0.4, 0.4, 0.4),
  });
  y -= lineHeight * 1.5;

  for (const field of fields) {
    const value = data[field.id] ?? data[field.name] ?? '';
    if (y < margin + lineHeight * 2) {
      page = doc.addPage([pageWidth, pageHeight]);
      y = pageHeight - margin - 20;
    }
    drawText(`${field.name}:`, { bold: true });
    drawText(value || '(empty)', { indent: 16 });
    y -= 4;
  }

  const bytes = await doc.save();
  return Buffer.from(bytes);
}

function wrapText(
  text: string,
  font: { widthOfTextAtSize: (t: string, size: number) => number },
  size: number,
  maxWidth: number
): string[] {
  const lines: string[] = [];
  const words = text.split(/\s+/);
  let current = '';
  for (const w of words) {
    const next = current ? `${current} ${w}` : w;
    if (font.widthOfTextAtSize(next, size) <= maxWidth) {
      current = next;
    } else {
      if (current) lines.push(current);
      current = w;
    }
  }
  if (current) lines.push(current);
```

```
  return lines;
}
```