**Project overview (one paragraph)**

This app takes raw PDF extraction output (or a JSON file you already have), turns it into a consistent “store” structure (Document + Sections), and then loads that structure into a Neo4j graph so you can explore relationships like “this document has these sections,” “this section is a child of that section,” and “this section comes next.”

**What each folder/file does**

**main.py**

* Starts the FastAPI web server.
* Registers the API routers:
  + the extraction routes (normalize raw data into the store format),
  + the knowledge graph (KG) routes (load the store into Neo4j).
* Provides a simple health check and a welcome message.

**app/core/config.py**

* Reads settings and secrets (e.g., Neo4j URI, user, password, database).
* Makes these values available as settings everywhere else.
* Has a convenience flag neo4j\_enabled so routes can quickly check whether Neo4j is configured.

**app/routers/extraction.py *(your extraction endpoints)***

* **Endpoints to:**
  + Extract text from PDFs using different libraries.
  + Accept raw JSON (file upload or request body).
  + Normalize that input to the “store” format using the builder.
  + Optionally include a dynamic JSON Schema describing the store.
  + If asked (auto\_load\_to\_kg=true), call the KG importer to push the result into Neo4j.

**app/routers/kg.py *(your knowledge graph endpoints)***

* Endpoint to ensure Neo4j constraints (unique IDs for Document and Section).
* Endpoint to import a store into Neo4j (creates nodes and relationships).
* These endpoints are also used internally by the extraction routes when you set auto\_load\_to\_kg=true.

**app/services/pdf\_processor.py**

* Two helpers for extracting content from PDFs:
  + One uses PyMuPDF (fast plain text per page).
  + One uses Unstructured (richer element objects).
* The Unstructured import is lazy on purpose so the server still starts even if those extra packages are not installed.

**app/services/loaders.py**

* Accepts many JSON shapes and finds the array of “elements,” no matter whether it lives under elements, data, pages, or items.
* Used by the extraction routes to normalize incoming JSON before building the store.

**app/services/text.py**

* Looks inside each raw element and tries multiple keys/paths to find real text (e.g., text, content, metadata.text).
* Cleans and de-duplicates the text candidates.
* Returns a “best” text plus diagnostics (where it came from and other candidate strings).

**app/services/parsers.py**

* Simple rules to interpret headings and labels:
  + Detects things like ARTICLE I, Section 2.1, Exhibit A.
  + Extracts label, title, and an estimated level (depth in the hierarchy).
* Detects cross-references and definition terms in text using regular expressions.

**app/services/builder.py *(the heart of normalization)***

* Takes the list of raw elements and builds a Store object:
  + Assigns stable IDs using hashes (document → sections).
  + Creates sections with canonical text, page info, label/title/level, and diagnostics.
  + Detects cross-references (e.g., “See Section 2.1”) and tries to resolve them.
  + Detects definitions (capitalized terms in quotes with their sentences).
  + Prepares a light topology for quick navigation:
    - a map of parent → children,
    - and a section index with either text or a snippet/hash/length.
  + Writes provenance (who extracted, when, how many elements).
* Returns a Pydantic model that can be converted to plain JSON.

**app/services/kg.py *(Neo4j integration)***

* Opens a connection to Neo4j Aura.
* Ensures constraints so you don’t create duplicates:
  + unique doc\_id for Document,
  + unique section\_id for Section.
* Converts the store into flat properties Neo4j accepts.
* Writes:
  + one Document node,
  + many Section nodes,
  + relationships:
    - HAS\_SECTION (document → each section),
    - PARENT\_SECTION (hierarchy),
    - NEXT\_SECTION (reading order).
* Uses batched Cypher so large imports are stable and fast.

**app/models/store.py *(data shapes)***

* Pydantic models describing:
  + Store (the entire normalized payload),
  + DocumentHeader, Section, Definition, CrossRef,
  + StoreBundle (what the API returns: store plus optional JSON schema).

**app/schemas/json\_schemas.py**

* Looks at one example store and infers a JSON Schema for it.
* Marks core fields as required but allows extra properties so the model stays flexible.

**app/utils/ids.py**

* Utility functions:
  + generate SHA-256 hashes,
  + build URN-style IDs,
  + get a UTC timestamp in ISO format.

**app/cli/build\_store.py *(optional command-line tool)***

* Lets you run normalization from the terminal (not the API).
* Reads a JSON file, builds the store, writes the store to disk, and also writes the dynamic JSON Schema.

**The end-to-end flow**

1. **You send data to the API**
   * **Either upload a JSON file (already extracted),**
   * **or upload a PDF and let the service extract elements,**
   * **or send a raw JSON body.**
2. **Normalize**
   * **The loader finds the elements.**
   * **The builder converts them into a standardized store: one Document, many Sections, with hierarchy and order.**
3. **(Optional) Schema**
   * **The dynamic JSON Schema is generated from that store so downstream systems know the shape of the data.**
4. **(Optional) Load to Neo4j**
   * **If enabled, the KG service creates the Document and Section nodes, plus the relationships, inside Neo4j Aura.**
5. **Explore**
   * **In Neo4j Browser or Bloom, you can now query and visualize the document’s structure and section connections.**

**Why the graph looks the way it does**

* **Document and Section are the only node types you need to navigate any contract.**
* **HAS\_SECTION shows membership, PARENT\_SECTION shows hierarchy, NEXT\_SECTION shows order.**
* **This minimal shape keeps the system simple now, but you can extend it later with new node types (Definitions, Parties, Dates) without breaking anything.**

**How to run it (short)**

1. **Configure Neo4j credentials in .env (URI, user, password, database).**
2. **Install dependencies from requirements.txt.**
3. **Start the server with Uvicorn.**
4. **Use Swagger UI (/docs) to try:**
   * **Extraction routes to build the store,**
   * **KG routes to ensure constraints and import the store.**

**Common issues and what they mean**

* **“Neo4j not configured”  
  The Neo4j URI wasn’t loaded; check .env or environment variables.**
* **Import hangs  
  Large store writing to Aura; the batch import inside KG service handles this, but very big files still take time.**
* **Unstructured import error  
  Extra PDF dependencies aren’t installed; install unstructured[pdf] (or the listed packages individually).**

**How to extend later (without breaking)**

* **Add new node/edge types for things you care about (e.g., Definition, Party, REFERS\_TO).**
* **Keep changes additive and bump the schema version (e.g., 1.1.0).**
* **Update the KG service to write the new nodes/edges.**
* **Document the new queries your team can run.**

**QUESTIONS**

**1) How I came up with the schema**

**Goal first, not fields.**  
I asked: *“What’s the minimum structure we need to navigate and analyze an M&A agreement?”*  
Answer: the **document**, its **sections**, and how sections are **organized** and **ordered**.

So I modeled only:

* **Nodes**
  + Document — the whole file
  + Section — any heading/clause/paragraph with text
* **Relationships**
  + HAS\_SECTION — membership (document → section)
  + PARENT\_SECTION — hierarchy (section → parent section)
  + NEXT\_SECTION — reading order (section → next section)

This is the **smallest useful graph** that still lets us do real analysis (drill-down, compare order, see hierarchy).

**2) Why “only this much” schema was required**

* **Keep it useful and stable.** Complex data models break early; a small model works with every document.
* **All text preserved.** Even if a field isn’t in the schema, the **full text** of each section is kept; nothing is lost.
* **Metadata is flexible.** We retain per-section properties like label, title, page\_start, element\_type. If more metadata appears, the store already allows it (we don’t block extra properties).
* **Future-proof.** It’s easy to add new node types later (e.g., Party, Definition, Date, Obligation) without changing what already works.

Think of it as **Phase 1**: structure and navigation first. Deeper semantics come in Phase 2+.

**3) What if we find new important things that aren’t in the schema?**

We follow a **deliberate, low-risk extension path**:

1. **Discover**
   * We see new patterns in documents (e.g., defined terms, party names, effective dates, cross-references).
   * We log what we’re missing.
2. **Prototype on a branch**
   * Add *derived nodes* and relationships:
     + (:Definition {term, text}) + (:Section)-[:DEFINES]->(:Definition)
     + (:Section)-[:REFERS\_TO]->(:Section) for cross-references
     + (:Party {name}) + (:Document)-[:INVOLVES]->(:Party)
     + (:Date {value, kind}) + (:Section)-[:MENTIONS]->(:Date)
3. **Evaluate**
   * Does this help answer real questions (search, review time, traceability)?
   * Cost vs. benefit (storage, runtime, complexity).
4. **Adopt with versioning**
   * Bump **schema\_version** (e.g., 1.1.0) and keep changes **additive** (no breaking changes).
   * Write a migration/import step for legacy data if needed.
   * Document the new fields/labels with examples.
5. **Monitor**
   * Add dashboards/queries to detect empty/low-quality fields or schema drift.
   * Iterate.

This way, the schema **evolves with evidence**, not guesses.

**4) How the “dynamic JSON Schema” fits in**

* We **infer** a JSON Schema from actual data produced by the builder (types and required keys like doc\_id, section\_id, text).
* It’s **dynamic** so it adjusts to the real payload shape and doesn’t over-constrain.
* We keep additionalProperties: true in the right places so **new metadata can appear** without breaking validation.
* The schema is meant for **validation + documentation**, not to freeze the model forever.

**5) “why this schema?”**

**Short answer:**

“We designed a **minimal, document-first graph**: Document → Sections, with hierarchy and order. It’s the smallest model that lets us search, navigate, and analyze M&A contracts quickly. We kept it flexible so we can add Definitions, Parties, Dates, and Cross-Refs later without breaking anything.”

**If they ask about missing fields:**

“Nothing is lost — all section text is stored. When we identify a new concept we need (like Parties or Definitions), we add it as new node/edge types, bump the schema version, and keep the old data valid.”

**If they ask how you’ll know when to extend:**

“We extend only when a new question can’t be answered with the current graph, or when we see repeated patterns across documents. We prototype, measure the benefit, then adopt with versioning.”

**6) Practical criteria for promoting new concepts to the schema**

Promote something to a **first-class node/edge** when:

* It’s **recurrent** across many docs (not a one-off).
* It materially improves **findability** or **explainability** (e.g., ‘all termination triggers’).
* It reduces **review time** or **error risk**.
* It’s **extractable** with acceptable precision/recall.

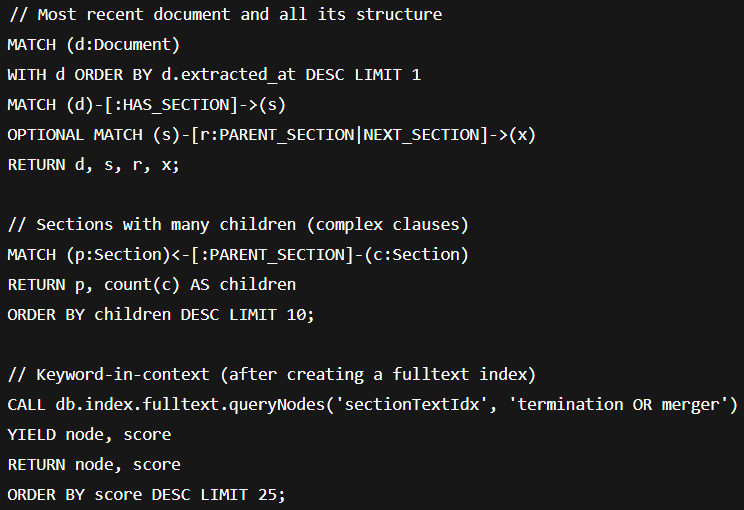
Examples:

* **High fit:** Definition, Party, Date, Money, Jurisdiction, Obligation, Condition.
* **Keep as properties for now:** one-off flags, page-specific hints, extractor diagnostics.

**7) How to analyze this graph today (concrete tips)**

* **Structure:** count sections per level; find largest subtrees.
* **Order:** follow NEXT\_SECTION to understand flow (e.g., what comes before/after “Termination”).
* **Search:** full-text index on Section.text to quickly find clauses, then pivot to neighbors (parent/children/next).
* **Compare documents:** load two docs and compare subtree sizes or the presence of specific headings.

Useful Cypher starters:



**8) What if we realize we need more *now*?**

Plan a **Phase 2**:

* Add:
  + Definition nodes + DEFINES
  + REFERS\_TO edges (resolved cross-references)
  + Party nodes + INVOLVES
  + Date nodes + MENTIONS
* Update the importer to populate them (all **additive**).
* Bump schema\_version → 1.1.0.
* Ship queries and dashboards that prove the value.