**0) You open the web page**

* **Request**: GET /
* **Handled by**: app/main.py → index()
  + Reads and returns frontend/index.html.
* **Static assets**:
  + frontend/index.html references /static/styles.css and /static/app.js.
  + These are served by FastAPI’s static mount: app.mount("/static", ...) in app/main.py.
* **What the page does on load**:
  + frontend/app.js immediately calls **createNewSession()** (bottom of the file).
    - **Request**: POST /session/new
    - **Handled by**: app/main.py → new\_session()
      * Generates a session\_id (UUID) and returns it as JSON.
    - The JS stores session\_id in currentSessionId and renders it in the Session box.

Why a session first?  
So all documents you upload and all queries you make are *scoped* to this session\_id, keeping results consistent and isolated.

**1) You upload one or more files**

* **UI action**: Choose files → click **Upload** (in “1) Upload documents”).
* **File**: frontend/app.js → uploadForm.addEventListener('submit', …)
  1. It ensures a session exists (calls createNewSession() if missing).
  2. It blocks zero-byte files.
  3. It builds FormData with:
     + files (one or many)
     + session\_id (the current one)
  4. **Request**: POST /upload (multipart/form-data)
* **Backend endpoint**: app/main.py → upload(...)
  1. Loops over each UploadFile:
     + Reads bytes into memory (for diagnostics: length, first bytes, sha256).
     + Calls **ingest\_file(...)** (core ingestion) with session\_id.
* **Core ingestion pipeline**: app/ingest.py → ingest\_file(file\_bytes, filename, session\_id=...)
  1. **save\_upload**: Writes the raw file bytes to disk under data/uploads/<filename>.
  2. **extract\_text**:
     + Branches on extension:
       - **PDF**:
         * parse\_pdf\_text\_layer (via **PyMuPDF**) tries to read the text layer.
         * If empty, **OCR fallback**: ocr\_pdf\_pages (PyMuPDF renders → Pillow image → **pytesseract** OCR).
       - **DOCX**: parse\_docx (**python-docx**) extracts paragraphs and tables.
       - **TXT/MD**: parse\_txt opens bytes and decodes with robust heuristics (BOMs, UTF-16/32, cp1252, latin-1, etc.).
  3. If still no text → raises a clear ValueError (the /upload endpoint captures this and reports status: "error" for that file).
  4. **Chunking**: chunk\_text (from app/rag.py) splits text into ~1000-char chunks with 200 overlap (paragraph/sentence aware).
  5. For each chunk, builds a Chunk dataclass (id, text, metadata). Metadata includes:
     + filename, document\_id, chunk\_index, ext, used\_ocr, and **session\_id**
  6. **Vectorization + upsert**: upsert\_chunks:
     + Calls embed\_texts (**SentenceTransformers** with sentence-transformers/bert-base-nli-mean-tokens) to embed all chunk texts.
     + Upserts vectors into **ChromaDB** persistent collection (“docs”), under storage/chroma/.
  7. Returns JSON summary { document\_id, filename, chunks, used\_ocr, encoding, session\_id }.
* **Backend response**: app/main.py → upload(...)  
  Returns a per-file array with {filename, status: "ok"/"error", detail, diag}.
* **UI renders**: frontend/app.js prints the JSON in the Upload panel.

At this point, your session contains vectors for the uploaded documents. You can upload more files; they all go into the same session\_id namespace.

**2) You write a system prompt (optional)**

* **UI**: Textarea “2) System Prompt (optional)”.
* Stored only on the client until you ask a question; it’s sent with the /ask payload to steer the answer (tone/constraints).

**3) You ask a question**

* **UI action**: Type question → click **Ask**.
* **File**: frontend/app.js → askBtn.addEventListener('click', …)
  1. Ensures currentSessionId exists.
  2. **Request**: POST /ask (JSON payload):
  3. {
  4. "question": "...",
  5. "system\_prompt": "...", // optional
  6. "top\_k": 8,
  7. "session\_id": "<uuid>"
  8. }
* **Backend endpoint**: app/main.py → ask(payload: AskRequest)
  1. Calls rag\_ask(payload.question, payload.system\_prompt, payload.top\_k, payload.session\_id or "default").
* **RAG pipeline**: app/rag.py
  1. **ask(...)**:
     + similarity\_search(query, top\_k, session\_id)
       - **Embeds** the query (same embedding model).
       - **Filters by session**: where={"session\_id": session\_id} so only vectors for your current session are considered.
       - **Chroma query** (cosine space) returns top-K candidate chunks (text + metadata).
     + rerank(query, passages, top\_k=min(5, top\_k))
       - **Cross-encoder** (cross-encoder/ms-marco-MiniLM-L-6-v2) re-scores each (query, passage) pair for relevance.
       - Sorts by rerank score and truncates to top ~5.
     + build\_prompt(system\_prompt, question, contexts)
       - Concatenates the best chunk texts into a **Context** section.
       - Prepends your **System Prompt** (or a safe default instructing “answer from context only”).
       - Appends your **Question**.
     + generate\_answer(system\_prompt, question, contexts)
       - Uses **FLAN-T5** (google/flan-t5-base) with deterministic decoding (beam search) to produce the **final answer** from the assembled prompt.
     + Returns (answer, sources) where sources are the top reranked chunks (id, score, text excerpt, metadata including filename and session).
  2. **Models involved** (already warm if previously used in this process):
     + Embeddings: sentence-transformers/bert-base-nli-mean-tokens
     + Re-ranker: cross-encoder/ms-marco-MiniLM-L-6-v2
     + Generator: google/flan-t5-base
* **Backend response**:  
  AskResponse(answer: str, sources: List[dict])
* **UI renders**: frontend/app.js → renderSources(json.sources)  
  Displays the answer and an expandable list of the supporting chunks, including filename, rerank score, and session id.

**4) You start over with a clean set of docs (two ways)**

**A) New Session (recommended)**

* **UI action**: Click **New Session**.
* **File**: frontend/app.js → newSessionBtn.addEventListener('click', …)
  + **Request**: POST /session/new
  + **Backend**: app/main.py → new\_session() returns a new session\_id.
  + The UI switches to that new session\_id and clears the panels.
  + Upload docs again → everything you ask is restricted to this new session’s vectors.

**B) Reset Session (delete vectors for current session)**

* **UI action**: Click **Reset Session**.
* **File**: frontend/app.js → resetSessionBtn.addEventListener('click', …)
  + **Request**: POST /session/reset (form-encoded session\_id)
  + **Backend**: app/main.py → reset\_session(session\_id) calls:
    - app/rag.py → delete\_session(session\_id)  
      which issues a Chroma delete with where={"session\_id": session\_id} (removes all vectors in that namespace).
  + The UI clears the current results; you can upload a fresh set into the same session\_id.

Note: the original files you uploaded stay on disk under data/uploads/. Resetting a session clears the **vector entries** for that session in storage/chroma/. If you re-upload the same physical file, it will be re-embedded.

**5) Where data and models live**

* **Uploads**: data/uploads/<filename> (raw bytes saved at ingest)
* **Vector store**: storage/chroma/ (ChromaDB persistent index)
* **Model cache** (first run only):  
  Windows: C:\Users\<you>\.cache\huggingface\hub\  
  Linux/macOS: ~/.cache/huggingface/hub/

**6) Error handling you’ll see**

* /upload returns {results: [ {filename, status: "ok"/"error", detail, diag} ]} for **each file**; one bad file won’t break a batch.
* Common messages:
  + **TXT**: “No extractable text…” + hint to save as UTF-8 (we already try many encodings; most issues are empty files).
  + **PDF**: If it’s scanned and Tesseract isn’t installed, you’ll get a hint to install it. If installed, used\_ocr: true appears.
* The UI prints the JSON so you can see which files were ingested and how (including encoding, used\_ocr, session\_id).

**7) Why consistency improves now**

* Every chunk written to Chroma includes **session\_id** in its metadata.
* Every POST /ask includes that same **session\_id**.
* Retrieval (similarity\_search) adds where={"session\_id": <id>}, so you never “leak” context across sessions.
* **New Session** gives you a clean namespace. **Reset Session** wipes the namespace.

**8) Minimal call graph (at a glance)**

**Page load**

GET / → app/main.py:index() → frontend/index.html

index.html → /static/styles.css, /static/app.js

app.js:init() → POST /session/new → new\_session() → {session\_id}

**Upload**

app.js:submit upload → POST /upload (files[], session\_id)

app/main.py:upload() → ingest\_file() [app/ingest.py]

→ save\_upload()

→ extract\_text()

↳ parse\_pdf\_text\_layer() | ocr\_pdf\_pages() | parse\_docx() | parse\_txt()

→ chunk\_text() [app/rag.py]

→ upsert\_chunks() [app/rag.py]

↳ embed\_texts() [SentenceTransformer]

↳ chroma upsert (persisted)

**Ask**

app.js:Ask → POST /ask {question, system\_prompt, top\_k, session\_id}

app/main.py:ask() → rag\_ask() [app/rag.py]

→ similarity\_search() (where={"session\_id": ...}) → Chroma query

→ rerank() (CrossEncoder)

→ build\_prompt()

→ generate\_answer() (FLAN-T5)

→ return answer + sources

app.js:renderSources()

**Session controls**

New Session → POST /session/new → returns new session\_id

Reset Session → POST /session/reset {session\_id} → delete\_session() → Chroma delete where session\_id

If you want, I can also add:

* A **document list** per session (with per-document delete),
* A **session switcher** (dropdown of recent session\_ids),
* A **/health** endpoint that tests model availability and vector DB status.

But with the above, you’ve got a clean, deterministic workflow from page open → isolated uploads → deterministic RAG answers → easy reset.