Nice — this is a great hackathon problem. I’ve turned it into a concrete, 36-hour plan with an MVP, step-by-step roadmap, the open-source stack you’ll use, data models, API contracts, task assignments, and a demo checklist so you can build like a production team.

MVP (must-have for the 36-hour submission)

1. Document upload (PDF, DOCX, TXT) + store files locally and metadata in SQLite.
2. Automatic classification into the fixed categories (Finance, HR, Legal, Contracts, Technical Reports, Invoices).
3. Metadata extraction (title, author, date) + simple entity extraction (names, orgs, amounts) using spaCy / regex.
4. Extractive summarization (TF-IDF / sentence scoring) → top 3 sentences.
5. Semantic search: build embeddings with sentence-transformers, store in FAISS (in-memory) and search by cosine similarity.
6. Simple auth & role-based access (username/password in SQLite). HR sees HR docs only; Finance sees Finance docs only.
7. Frontend dashboard: upload, list documents, view metadata & summary, semantic search with filters.
8. Access logs (upload, view) stored in DB.

Overall tech stack (fastest, production-like)

- Backend: **FastAPI (Python)** — lightweight, async, easy to ship.
- Frontend: **Next.js + Tailwind CSS** (or plain React + Tailwind) — fast dashboard scaffold.
- DB: **SQLite** (for hackathon simplicity) — store metadata, users, logs.
- File storage: local filesystem (uploads/) during demo.
- NLP / ML libs: **spaCy**, **sentence-transformers**, **scikit-learn** (TF-IDF + LogisticRegression), **faiss-cpu**, **nltk** (optional).

- Document parsing: **PyPDF2** / **pdfminer.six** for PDFs, **python-docx** for DOCX, plain read for TXT.
- Auth: JWT cookies OR simple session cookies; password hashing with **bcrypt**.
- Dev / Deployment: GitHub (repo), Vercel for frontend, Railway/Render for backend (or run locally for demo).

Architecture (high level)

Frontend <--> FastAPI REST API <--> SQLite + files + FAISS (in memory)

- Worker/endpoint for: upload → parse → metadata extraction → classification → summarization → embedding → add to FAISS + DB.

Data model (SQLite tables)

users

- id, username, password_hash, role (admin/hr/finance/viewer), created_at

documents

- id, filename, filepath, uploader_id, category, title, author, date_extracted, summary (text), uploaded_at

entities

- id, document_id, entity_type (PERSON/ORG/AMOUNT/EMAIL), entity_text

embeddings

- doc_id, embedding_vector_filepath OR maintain FAISS index + mapping table {doc_id -> index}.

access_logs

- id, user_id, action (upload/view/search/download), doc_id (nullable), timestamp, ip

Simple file layout (repo)

```
/backend
  app/
    main.py (FastAPI)
    routes/
      auth.py
      docs.py
      search.py
    services/
      parser.py
      extractor.py
      classifier.py
      summarizer.py
      embeddings.py
    models.py (SQLAlchemy)
    db.py
/frontend
  pages/
    index.tsx
    login.tsx
    dashboard.tsx
    upload.tsx
    doc/[id].tsx
  components/
README.md
```

Key API endpoints (MVP)

- `POST /api/auth/login` — body: {username, password} → returns session cookie / JWT
- `POST /api/docs/upload` — multipart file + optional metadata → returns doc_id
- `GET /api/docs` — query params: category, author, date_from, date_to → returns list with Title, Category, Author, Date, doc_id
- `GET /api/docs/{id}` — metadata + summary + entities + download URL
- `GET /api/search?q=...&filters=...` — performs semantic search; returns ranked doc list with relevance score
- `GET /api/logs` — admin/forensics (view logs)

Algorithms & implementation details (concrete)

1. **Parsing**
    - PDF: use `pdfminer.six` or `PyPDF2` to get raw text.
    - DOCX: `python-docx`.
    - TXT: read directly.
2. **Title extraction**
    - Heuristic: first bold line or the first non-empty line > 5 chars. If text formatting unavailable, choose first sentence or first line with capitalized words. (Implement fallback logic.)
3. **Author extraction**
    - Regex search for "Author:", "By:", "Submitted by", and look for email patterns `\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b`. spaCy can help find PERSON entities as fallback.
4. **Date extraction**
    - Regex patterns for dd/mm/yyyy, yyyy-mm-dd, Month names. Use `dateparser` if available or a basic regex + `datetime.strptime` fallbacks.
5. **Entity extraction**
    - Run spaCy NER (`en_core_web_sm` or `en_core_web_trf` if resources permit) and capture PERSON, ORG, MONEY, DATE, EMAIL (via regex), etc.
6. **Summarization (extractive)**
    - Sentence tokenize (nltk / spacy). Compute TF-IDF over sentences (scikit-learn `TfidfVectorizer`) or simple word frequency scoring. Score each sentence and select top 3–5 sentences preserving document order.
7. **Classification** (MVP approach)
    - Option A (fast/robust): TF-IDF on document text → LogisticRegression or RandomForest trained on small seed dataset (you can bootstrap with a few sample documents per category). This is fast to implement.

    - Option B (stretch): Use sentence-transformers to embed doc and train a simple classifier in embedding space or use nearest-centroid per category.

8. **Embeddings & Semantic Search**
    - Use `sentence-transformers` (e.g., `all-MiniLM-L6-v2`) to generate vector for whole document (or for sections).
    - Build FAISS index: `faiss.IndexFlatIP` (with normalized vectors) or `IndexFlatL2`. Keep a mapping `doc_id -> index_pos`. For persistence, save FAISS index to disk after each update.

9. **Ranking**
    - On search: embed query → search FAISS → get top-k doc ids → re-rank optionally by combining cosine score and metadata matches (e.g., category filter).

Security & privacy notes

- Run all models locally or in your own backend environment — **do not** call external proprietary LLMs for sensitive docs (unless permitted).
- Hash passwords with bcrypt; use HTTPS when deploying.
- Enforce RBAC on every endpoint; confirm user role before returning results.
- Limit file upload size and scan for malicious files (basic validation by extension + MIME type).
- Store sensitive files encrypted at rest as a stretch goal (outside hackathon scope).

36-Hour sprint plan (detailed)

Hour 0–2 — Kickoff & scaffold

- Create repo, branches, README, .gitignore.
- Scaffold FastAPI app + Next.js app, simple Dockerfile (optional).
- Create DB models and migrations (or simple create tables script).
- Quick UI mockup (wireframes).

Hour 2–8 — Document ingestion pipeline (MVP core)

- Implement file upload route, save file, insert metadata skeleton in SQLite.
- Implement parsers for PDF/DOCX/TXT and store raw text.
- Implement title/author/date heuristics.
- Add access log entry for upload.

Hour 8–14 — Summarization + Entities

- Integrate spaCy; extract entities (PERSON/ORG/MONEY).
- Implement extractive summarizer (TF-IDF sentence scoring).
- Save summary & entities to DB.

Hour 14–20 — Classification

- Implement TF-IDF vectorizer + LogisticRegression classifier.
- Seed small training dataset (create 10–20 small example docs per category or create synthetic examples).
- Run classification pipeline and store category.

Hour 20–26 — Embeddings & Semantic Search

- Integrate `sentence-transformers` to get doc embeddings.
- Create FAISS index, insert embeddings, store mapping.
- Implement `/api/search` to return ranked results.
- Add filters (category, author, date).

Hour 26–30 — Auth + RBAC + Logs

- Implement login, roles, route guards.
- Ensure `/api/docs` filters returned docs based on user role.
- Store view/download logs.

Hour 30–34 — Frontend wiring + polish

- Implement upload form, documents list, document view page (metadata + summary + download).
- Implement search bar with filters and result ranking display.
- Simple responsive CSS; show role-based filtered lists.

Hour 34–36 — Testing, demo prep, README, deploy (if possible)

- Fix bugs, add instructions for running locally, prepare sample dataset.
- Prepare demo script and 1-minute pitch.

- If time: push backend to Railway and frontend to Vercel and demonstrate live.

Role assignments & tasks (for a team of 3–4)

- **Frontend dev (1)**
  - Implement Next.js pages (login, dashboard, upload, doc view, search).
  - Integrate API calls + token storage.
  - Polishing & demo UI.
- **Backend dev (1–2)**
  - FastAPI routes (auth, upload, docs, search).
  - DB models, file handling, access logs.
  - Deploy backend.
- **ML / NLP dev (1)**
  - Parsers, metadata extraction, spaCy pipeline.
  - Summarization, classifier training, embeddings + FAISS.
  - Provide model artifacts and ensure fast inference.
- **DevOps / QA (shared)**
  - Docker config, CI basic workflow, deploy steps, test flows, demo readiness.

Starter open-source resources & libs (no external links but exact names)

- FastAPI, Uvicorn
- Next.js, React, Tailwind CSS

- spaCy (en_core_web_sm)
- sentence-transformers (Hugging Face models like all-MiniLM-L6-v2)
- FAISS (faiss-cpu)
- scikit-learn (TfidfVectorizer, LogisticRegression)
- PyPDF2 / pdfminer.six / python-docx
- bcrypt, python-jose (JWT)
- SQLite (sqlite3) or SQLAlchemy for ORM
- NLTK (sentence tokenization) or spaCy for sentences
- dateparser (optional) for robust date extraction

Minimal sample pseudocode (classification + embed + faiss insertion)

```
# after parsing text -> raw_text
# classification (MVP)
X = tfidf_vectorizer.transform([raw_text])
category = clf.predict(X)[0]

# embedding & faiss
vec = embed_model.encode([raw_text], normalize_embeddings=True)
idx = faiss_index.add(vec)
doc_index_map[doc_id] = current_faiss_index_pos
```

Testing checklist (quick)

- Upload PDF/DOCX/TXT -> parsed text stored.
- Title/Author/Date extracted for multiple formatting patterns.
- Classification returns a category for every upload.
- Summary returns 3 sentences that are coherent.
- Search returns relevant documents for semantic queries.

- RBAC: HR user cannot access Finance docs.
- Logs record uploads & views.

Demo script (2–3 minutes)

1. Login as Admin / HR / Finance (show role effect).
2. Upload one Finance doc and one HR doc (show upload logs).
3. Show that metadata (title, author, date) and summary are extracted & visible.
4. Use semantic search: query a phrase (e.g., “employee salary reimbursement”) → show Finance doc returned ranked top.
5. Show access control: login as HR user and show only HR docs.
6. End with architecture slide + highlight of privacy (models run locally) and next steps (index persistence, encrypted storage, improved classifier).

Success metrics (what judges will look for)

- Functionality: end-to-end upload→classify→search works.
- UX: easy-to-use dashboard and clear role-based behavior.
- Accuracy: summaries & extracted metadata are readable and correct most of the time.
- Privacy: all processing done server-side; no external API calls.
- Scalability: FAISS & embedding approach is extensible.

Stretch features (if time permits)

- Persist FAISS index to disk and reload on startup.
- Use a small fine-tuned transformer classifier from Hugging Face for better accuracy.
- UI: preview first page of PDF, show highlighted sentences used in summary.
- Audit logs UI & export logs.
- Encryption at rest for files.

- Add incremental ingestion and background worker (Celery/RQ) for heavy files.

If you want, I can immediately:

- generate the **FastAPI** skeleton (routes + models) and a minimal `classifier.py` + `embeddings.py`, or
- scaffold the **Next.js** dashboard pages (login, upload, docs list, doc view).

Tell me which scaffolding to produce first (backend FastAPI skeleton or frontend Next.js starter) and I'll produce ready-to-copy code next.