✅ WEEK 20 FINAL PROJECT — Domain-Specific Q&A Bot (Step-by-Step)

What you will build

A bot where the user types a question like:

> "How many marks is the final exam worth?"

Your system will:

1. search your PDF/notes
2. pick the best relevant chunks
3. give an answer **with sources** like:

> "Answer based on syllabus.pdf, page 4"

---

0) Choose the best project domain (fast decision)

Best for beginners + portfolio

✅ **Course syllabus / class notes / textbook Q&A**

Why: clean, small docs, easy evaluation, easy demo.

If you want "professional / corporate"

✅ **Company policy bot**

Why: shows real-world usefulness (HR/IT policies).

If you want "research vibe"

✅ **Research paper bot**

Why: shows technical depth, but papers are harder (tables, formulas).

My recommendation (default): Start with **Syllabus/Notes bot**. It's the easiest to finish cleanly.

---

1) Create project folder structure

Make this structure:

```
week20_domain_qa_bot/
  data/
    docs/             # put PDFs or txt here
  chroma_store/       # vector db saved here
  src/
    ingest.py
    rag_chain.py
    ui_app.py
    eval.py
  README.md
  requirements.txt
```

---

2) Install libraries

2.1 Create requirements.txt

Put this inside requirements.txt:

```
langchain
langchain-core
langchain-community
langchain-openai
langchain-chroma
chromadb
pypdf
python-dotenv
gradio
```

2.2 Install

```
pip install -r requirements.txt
```

---

3) Add your documents

Put your files here:

- data/docs/syllabus.pdf

- data/docs/notes.pdf
- data/docs/policy.pdf (optional)

Tip: Start with 1 PDF first. Add more later.

---

4) Ingestion Step (Load PDF → chunk → store in vector DB)

Create: src/ingest.py

✅ What this does

- Reads PDFs
- Breaks them into small chunks
- Saves them into **ChromaDB** (vector DB) with metadata like page number

Code (copy-paste)

```
import os
from pathlib import Path
from pypdf import PdfReader

from langchain_core.documents import Document
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings


DOCS_DIR = Path("data/docs")
CHROMA_PATH = "chroma_store"
COLLECTION_NAME = "week20_domain_qa"


def load_pdf_as_documents(pdf_path: Path):
    reader = PdfReader(str(pdf_path))
    docs = []

    for i, page in enumerate(reader.pages):
        text = page.extract_text() or ""
        text = " ".join(text.split())  # light cleaning
        if text.strip():
            docs.append(
                Document(
                    page_content=text,
                    metadata={
```

```
                    "source": pdf_path.name,
                    "page": i + 1
                }
            )
        )
    return docs


def chunk_documents(docs, chunk_size=350, overlap=80):
    """
    Simple word-based chunking.
    chunk_size=350 words, overlap=80 words.
    """
    chunked = []
    for d in docs:
        words = d.page_content.split()
        start = 0
        chunk_id = 0

        while start < len(words):
            end = start + chunk_size
            chunk_text = " ".join(words[start:end])

            chunked.append(
                Document(
                    page_content=chunk_text,
                    metadata={
                        **d.metadata,
                        "chunk_id": chunk_id
                    }
                )
            )

            chunk_id += 1
            start = end - overlap

    return chunked


def build_vector_db():
    embeddings = OpenAIEmbeddings()  # uses your OpenAI key

    # Load all PDFs
    all_docs = []
    for file in DOCS_DIR.glob("*.pdf"):
        all_docs.extend(load_pdf_as_documents(file))

    # Chunk them
    chunked_docs = chunk_documents(all_docs)

    # Create / persist ChromaDB
    db = Chroma(
        collection_name=COLLECTION_NAME,
        embedding_function=embeddings,
        persist_directory=CHROMA_PATH
    )

    # Add to DB
    db.add_documents(chunked_docs)

    print(f"✅ Loaded {len(all_docs)} pages and stored {len(chunked_docs)} chunks in
ChromaDB.")


if __name__ == "__main__":
    build_vector_db()
```

Run it

```
python src/ingest.py
```

✅ After this, your vector DB is ready and saved in chroma_store/.

---

5) Build the RAG pipeline (Retriever → Prompt → LLM → Answer + Sources)

Create: src/rag_chain.py

✅ What this does

- Takes a question
- Retrieves top chunks
- Makes a strict prompt ("use context only")
- Returns answer + citations

```
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser


CHROMA_PATH = "chroma_store"
```

```
COLLECTION_NAME = "week20_domain_qa"


def format_docs(docs):
    """
    Convert retrieved docs into citation-friendly context.
    """
    blocks = []
    for d in docs:
        src = d.metadata.get("source", "unknown")
        page = d.metadata.get("page", "NA")
        chunk_id = d.metadata.get("chunk_id", "NA")

        blocks.append(
            f"[Source: {src} | Page: {page} | Chunk: {chunk_id}]\n{d.page_content}"
        )
    return "\n\n".join(blocks)


def get_rag_answer(question: str, top_k=5):
    embeddings = OpenAIEmbeddings()
    db = Chroma(
        collection_name=COLLECTION_NAME,
        embedding_function=embeddings,
        persist_directory=CHROMA_PATH
    )

    retriever = db.as_retriever(search_kwargs={"k": top_k})

    # Retrieve docs first (so we can also compute confidence later)
    docs = retriever.invoke(question)
    context = format_docs(docs)

    prompt = ChatPromptTemplate.from_messages([
        ("system",
         "You are a helpful assistant. "
         "Answer ONLY using the provided context. "
         "If the answer is not in the context, say: 'I don't know.' "
         "At the end, list the sources you used (source + page)."),
        ("user",
         "Context:\n{context}\n\nQuestion:\n{question}\n\nAnswer:")
    ])

    llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
    chain = prompt | llm | StrOutputParser()

    answer = chain.invoke({"context": context, "question": question})
    return answer, docs
```

---

6) Add confidence-based responses (simple and practical)

We'll do a beginner-friendly "confidence" rule:

- If retrieved chunks are weak / irrelevant → bot should say **"I don't know"** or ask user to rephrase.

Simple confidence method

Use similarity score if available OR do a quick "keyword match check".

Easy approach (works everywhere):

- If none of the retrieved chunks contain even 1–2 important words from the question, confidence is low.

Add this to rag_chain.py:

```
def simple_confidence(question: str, docs):
    q_words = [w.lower() for w in question.split() if len(w) > 3]
    joined = " ".join([d.page_content.lower() for d in docs])

    hits = sum(1 for w in q_words if w in joined)
    # crude score: 0 to 1
    score = hits / max(1, len(q_words))

    return score
```

Then after retrieval:

- if score < 0.15 → "low confidence"

---

7) Build UI (Gradio is easiest)

Create: src/ui_app.py

```
import gradio as gr
from rag_chain import get_rag_answer, simple_confidence
```

```
def chat_fn(user_question):
    answer, docs = get_rag_answer(user_question, top_k=5)
    conf = simple_confidence(user_question, docs)

    if conf < 0.15:
        return "⚠️ I’m not confident because I couldn’t find strong evidence in your documents.
Try rephrasing or ask a simpler question."

    return f"{answer}\n\n(Confidence: {conf:.2f})"

demo = gr.Interface(
    fn=chat_fn,
    inputs=gr.Textbox(lines=2, placeholder="Ask something from your documents..."),
    outputs="text",
    title="Week 20 Domain Q&A Bot",
    description="Ask questions from your private PDFs. The bot answers with sources."
)

if __name__ == "__main__":
    demo.launch()
```

Run:

```
python src/ui_app.py
```

✅ You now have a working Q&A chatbot UI.

---

8) Compare “With RAG vs Without RAG” (deliverable requirement)

Create: src/eval.py

What you’ll do

- Prepare 10 questions (you know their answers are in your docs)
- Compare:
  - LLM alone
  - RAG pipeline
- Record which one is more accurate / less hallucination

Simple evaluation skeleton:

```
from langchain_openai import ChatOpenAI
from rag_chain import get_rag_answer
```

```
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

EVAL_QUESTIONS = [
    "What is the attendance requirement?",
    "How many marks is the final exam worth?",
    "What are the assessment components?"
]

def answer_without_rag(q):
    return llm.invoke(q).content

def run_eval():
    rows = []
    for q in EVAL_QUESTIONS:
        no_rag = answer_without_rag(q)
        rag, _ = get_rag_answer(q)

        rows.append({
            "question": q,
            "no_rag_answer": no_rag,
            "rag_answer": rag
        })

    for r in rows:
        print("\nQ:", r["question"])
        print("\n--- No RAG ---\n", r["no_rag_answer"])
        print("\n--- With RAG ---\n", r["rag_answer"])
        print("\n" + "="*60)

if __name__ == "__main__":
    run_eval()
```

✅ In your report, you'll write:

- where no-RAG guessed wrong
- where RAG used correct context + citations

---

9) Error analysis + limitations (what to write)

In 8–12 bullet points:

Common errors you’ll likely see

- PDF extraction misses tables
- chunk split breaks a key sentence
- retriever returns topic-related but not answer-rich text
- user asks vague questions
- “I don’t know” trigger too strict or too loose

Limitations (honest, looks professional)

- Small evaluation set
- Heuristic confidence scoring
- Not handling scanned PDFs (needs OCR)
- Doesn’t support images/tables well
- Needs reranking for best quality (optional upgrade)

---

10) README (portfolio-ready)

Your README must include:

10.1 Architecture diagram (simple ASCII)

```
User Question
  ↓
Chroma Retriever (top-k chunks)
  ↓
Context + citations
  ↓
LangChain Prompt Template
  ↓
LLM
  ↓
Answer + Sources + Confidence
```

10.2 How to run

1. put PDFs in data/docs/
2. run python src/ingest.py
3. run python src/ui_app.py

10.3 Demo questions

- “What is the final exam marks?”

- “What is attendance requirement?”

---

11) Optional upgrades (only if time)

If you want extra quality:

- Add **re-ranking** (CrossEncoder) before sending context to LLM (Week 18 Day 3 style)
- Add “source click” (if you store page mapping)
- Add multi-doc filters (“only syllabus.pdf”)

✅ Final checklist (submit-ready)

You should have:

- ✅ ingest.py builds vector DB
- ✅ rag_chain.py answers with citations
- ✅ ui_app.py provides chatbot UI
- ✅ eval.py shows with-RAG vs without-RAG
- ✅ README + limitations + results
