QUERY-VERSE

Tasks to do:
1. Backend:

Mongo DB (schema) [yash]
Firebase (authentiation) [jhan]-done
Redis [yash](in process skip)
Cloudinary [jhan]
Neo4j-done
Qdrant-done

Neo4j
uri ="neo4j+s://593eb7b1.databases.neo4j.io"
user = "neo4j"
password = "vwkSenzYtPp9bX6thdnJIU8BXXDm1WSfdqOIowYumRw"
connected = False

Quadrant
Endpoint:
https://90c18eba-c9f7-489f-9371-b46eea57639f.eu-central-1-0.aws.cloud.qdrant.io

```
from qdrant_client import QdrantClient

qdrant_client = QdrantClient(

url="https://90c18eba-c9f7-489f-9371-b46eea57639f.eu-central-1-0.aws.cloud.qdra
nt.io:6333",

api_key="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2Nlc3MiOiJtIn0.4UBgGu
a3TwRyiIlmpsJkdkD0spqfhfyr4xId3aASbOU",
)

print(qdrant_client.get_collections())
```

Backend :Firebase

Run this : npm install firebase

[mostly not required as it is script]

```
<script type="module">
  // Import the functions you need from the SDKs you need
  import { initializeApp } from "https://www.gstatic.com/firebasejs/11.6.0/firebase-app.js";
  import { getAnalytics } from
"https://www.gstatic.com/firebasejs/11.6.0/firebase-analytics.js";
  // TODO: Add SDKs for Firebase products that you want to use
  // https://firebase.google.com/docs/web/setup#available-libraries

  // Your web app's Firebase configuration
  // For Firebase JS SDK v7.20.0 and later, measurementId is optional
  const firebaseConfig = {
        apiKey: "AIzaSyAOKaSAg-fh1u_N1lEtnq8SiNdeRYLS1ek",
        authDomain: "faq-chatbot-9c17d.firebaseapp.com",
        projectId: "faq-chatbot-9c17d",
        storageBucket: "faq-chatbot-9c17d.firebasestorage.app",
        messagingSenderId: "210787549142",
        appId: "1:210787549142:web:bd0f835dd538e8d2f22e77",
        measurementId: "G-CK4NL92R85"
  };

  // Initialize Firebase
  const app = initializeApp(firebaseConfig);
  const analytics = getAnalytics(app);
```

</script>

Run this in your project directory: npm install -g firebase-tools

Deploy:[Do it in your project directory]

Run this:

firebase login

firebase init

firebase deploy[if deploying do this]

Cloudinary

Format:
CLOUDINARY_URL=cloudinary://<your_api_key>:<your_api_secret>@dhtjhhols

Actual:

CLOUDINARY_URL=cloudinary://182386449927386:gqSo9X2oyiIxnJQsxgYVSLhagJs
@dhtjhhols

Run: pip3 install cloudinary

File:main.py
```
import cloudinary
import cloudinary.uploader
from cloudinary.utils import cloudinary_url

# Configuration
cloudinary.config(
        cloud_name = "dhtjhhols",
        api_key = "182386449927386",
        api_secret = "gqSo9X2oyiIxnJQsxgYVSLhagJs", # Click 'View API Keys' above
to copy your API secret
        secure=True
)

# Upload an image
upload_result =
cloudinary.uploader.upload("https://res.cloudinary.com/demo/image/upload/getting-start
ed/shoes.jpg",
                                public_id="shoes")
print(upload_result["secure_url"])

# Optimize delivery by resizing and applying auto-format and auto-quality
optimize_url, _ = cloudinary_url("shoes", fetch_format="auto", quality="auto")
print(optimize_url)

# Transform the image: auto-crop to square aspect_ratio
auto_crop_url, _ = cloudinary_url("shoes", width=500, height=500, crop="auto",
gravity="auto")
print(auto_crop_url)
```

MongoDB

Name :jhansigonuguntla5

password:L0DWBgQBkjxPOMDF

[Take this only part]

mongodb+srv://atlas-sample-dataset-load-67f8d3e2cb9b0a5b87d753c0:L0DWBgQBkjxPOMDF@cluster1.bnroqjq.mongodb.net/

collection:cluster1.bnroqjq.mongodb.net

`mongodb-chatbot-schema.js`

```
// MongoDB Schema Design for FAQ Chatbot with Knowledge Retrieval

// User Collection
db.createCollection("users", {
  validator: {
            $jsonSchema: {
            bsonType: "object",
            required: ["username", "preferences", "createdAt"],
            properties: {
            username: {
            bsonType: "string",
            description: "Username must be a string and is required"
            },
            preferences: {
            bsonType: "object",
            properties: {
            theme: { bsonType: "string", enum: ["light", "dark", "system"] },
            notifications: { bsonType: "bool" },
            languagePreference: { bsonType: "string" }
            }
            },
            createdAt: { bsonType: "date" },
            lastLogin: { bsonType: "date" }
            }
            }
  }
});

// Sessions Collection - Tracking user interaction sessions
db.createCollection("sessions", {
  validator: {
            $jsonSchema: {
            bsonType: "object",
            required: ["userId", "startTime", "status"],
            properties: {
            userId: { bsonType: "objectId" },
            startTime: { bsonType: "date" },
            endTime: { bsonType: "date" },
            status: { bsonType: "string", enum: ["active", "closed"] },
            deviceInfo: { bsonType: "object" }
            }
            }
  }
});

// Chat Collection - Storing chat interactions
db.createCollection("chats", {
  validator: {
            $jsonSchema: {
            bsonType: "object",
            required: ["sessionId", "timestamp", "type"],
            properties: {
            sessionId: { bsonType: "objectId" },
            timestamp: { bsonType: "date" },
            type: { bsonType: "string", enum: ["query", "response", "feedback"] },
            content: { bsonType: "string" },
            metadata: {
            bsonType: "object",
            properties: {
            contextDocs: { bsonType: "array", items: { bsonType: "objectId" } },
            confidence: { bsonType: "double" },
            processingTime: { bsonType: "double" }
            }
            }
            }
            }
  }
});

// Documents Collection - Storing knowledge base documents
db.createCollection("documents", {
  validator: {
            $jsonSchema: {
            bsonType: "object",
            required: ["title", "content", "createdAt"],
            properties: {
            title: { bsonType: "string" },
            content: { bsonType: "string" },
            createdAt: { bsonType: "date" },
            updatedAt: { bsonType: "date" },
            category: { bsonType: "string" },
            tags: { bsonType: "array", items: { bsonType: "string" } },
            vector: { bsonType: "array", items: { bsonType: "double" } } // For vector search
capabilities
            }
            }
  }
});

// Sample functions to work with the collections

// Function to create a new chat session
function createChatSession(userId) {
  return db.sessions.insertOne({
            userId: ObjectId(userId),
            startTime: new Date(),
            status: "active",
            deviceInfo: { /* device information */ }
  });
}

// Function to add a query to chat
function addChatQuery(sessionId, queryText) {
  return db.chats.insertOne({
            sessionId: ObjectId(sessionId),
            timestamp: new Date(),
            type: "query",
            content: queryText
  });
}

// Function to add a chatbot response with context documents
function addChatResponse(sessionId, responseText, contextDocIds, confidence) {
  return db.chats.insertOne({
            sessionId: ObjectId(sessionId),
            timestamp: new Date(),
            type: "response",
            content: responseText,
            metadata: {
            contextDocs: contextDocIds.map(id => ObjectId(id)),
            confidence: confidence,
            processingTime: 235.5 // milliseconds
            }
  });
}

// Function to record user feedback
function addChatFeedback(sessionId, feedbackText, rating) {
  return db.chats.insertOne({
            sessionId: ObjectId(sessionId),
            timestamp: new Date(),
            type: "feedback",
            content: feedbackText,
            metadata: {
            rating: rating // e.g., 1-5 stars
            }
  });
}

// Function to retrieve recent chat history for a user
function getUserChatHistory(userId, limit = 10) {
  // First get the user's recent sessions
  const sessions = db.sessions.find({
            userId: ObjectId(userId)
  }).sort({ startTime: -1 }).limit(5).toArray();

  // Get chats for these sessions
  const sessionIds = sessions.map(session => session._id);
  return db.chats.find({
            sessionId: { $in: sessionIds }
  }).sort({ timestamp: -1 }).limit(limit).toArray();
}

// Create indexes for better performance
db.users.createIndex({ username: 1 }, { unique: true });
db.sessions.createIndex({ userId: 1 });
db.sessions.createIndex({ startTime: 1 });
db.chats.createIndex({ sessionId: 1 });
db.chats.createIndex({ timestamp: 1 });
db.documents.createIndex({ tags: 1 });
// Vector index for semantic search (requires MongoDB 5.0+ with Atlas Vector Search)
// db.documents.createIndex({ vector: "vector" }, { vectorOptions: { dimensions: 768,
similarity: "cosine" } });
```

```
// Example setup of aggregation pipeline for retrieving contextual information
function getContextualDocuments(query, limit = 3) {
  // This would typically use a vector search in production
  // For simplicity, we're using text search here
  return db.documents.find(
          { $text: { $search: query } },
          { score: { $meta: "textScore" } }
  )
  .sort({ score: { $meta: "textScore" } })
  .limit(limit)
  .toArray();
}
```