Make an app in RAG which should take document and according to that document it should answer user’s input  
  
frontend: vue.js

Backend: django

### Free google Gemini api key should be used (flash model) Report: Steps and Milestones for Developing a RAG App with Vue.js Frontend, Django Backend, and Google Gemini API

This report outlines the steps and milestones for developing a Retrieval-Augmented Generation (RAG) app. The app will allow users to upload a document, and based on that document, it will generate answers to user inputs. The frontend will be built using **Vue.js**, the backend will use **Django**, and the **Google Gemini API (Flash model)** will be utilized to process queries and generate responses.

**Milestone 1: Project Setup and Initial Planning**

**Steps:**

1. **Define Project Scope and Requirements**:
   * Define the functionality of the RAG app.
   * Clarify what kind of documents the app will handle (e.g., PDF, text files).
   * Specify the exact user input and expected output behavior (e.g., answering queries based on document content).
2. **Setup Version Control**:
   * Initialize a Git repository for the project (using GitHub, GitLab, etc.).
   * Set up a proper branch structure (e.g., development, main).

**Milestone 2: Frontend Development (Vue.js)**

**Steps:**

1. **Initialize Vue.js Project**:
   * Use Vue CLI to create a new Vue.js project.

bash

Copy code

vue create rag-app

* + Set up the basic folder structure for components, views, and assets.

1. **Design User Interface (UI)**:
   * Create pages for document upload and query input.
   * Build components for:
     + **Document upload**: A file input to allow users to upload documents.
     + **Query input**: A text input for users to enter questions.
     + **Results display**: A section to show the generated answers.
2. **Communicate with Backend**:
   * Use **Axios** (or another HTTP client) to send user queries and uploaded documents to the Django backend.

bash

Copy code

npm install axios

* + Set up the request-response cycle from the frontend to the backend.

1. **Add Basic Styling**:
   * Use CSS or libraries like **Bootstrap** or **Vuetify** for frontend styling.

**Milestone 3: Backend Development (Django)**

**Steps:**

1. **Set up Django Project**:
   * Create a new Django project and application.

bash

Copy code

django-admin startproject rag\_backend

cd rag\_backend

python manage.py startapp rag\_app

1. **Set Up Django REST Framework**:
   * Install **Django REST Framework** for building API endpoints.

bash

Copy code

pip install djangorestframework

* + Set up RESTful APIs to handle:
    - Document upload (file input).
    - Query handling (user input for questions).
    - Communicating with Google Gemini API.

1. **Handle Document Upload**:
   * Implement an API endpoint that accepts uploaded documents.
   * Use **Django’s FileField** or **FileUpload** to store the uploaded documents.
   * Parse the document (using libraries like **PyPDF2** for PDFs or **python-docx** for Word documents).
2. **API for User Queries**:
   * Create an API endpoint that accepts the user’s query and sends it to Google Gemini for processing.
   * Parse the response from Google Gemini and return it as a structured answer.

**Milestone 4: Integrating Google Gemini API (Flash Model)**

**Steps:**

1. **Obtain Google Gemini API Key**:
   * Sign up for Google Cloud and set up the **Google Gemini API** (Flash model).
   * Follow the steps to get the API key.
     + Google Cloud > AI Services > Gemini > Generate API key.
2. **Integrate API into Django Backend**:
   * Use the **requests** library to send POST requests to the Google Gemini API.

bash

Copy code

pip install requests

* + Send the user query (and document content if required) to the Gemini API and receive the answer.

1. **API Request Example**:

python

Copy code

import requests

def get\_answer\_from\_gemini(query, document):

url = 'https://gemini.googleapis.com/v1/models/flash:model'

headers = {

'Authorization': 'Bearer YOUR\_API\_KEY',

'Content-Type': 'application/json',

}

payload = {

"input\_text": query,

"context": document,

}

response = requests.post(url, headers=headers, json=payload)

return response.json()

1. **Process API Response**:
   * Extract and format the response from the Gemini API to present the answer to the user.
   * Handle potential errors (e.g., empty responses, timeouts, or API limits).

**Milestone 5: Connecting Frontend and Backend**

**Steps:**

1. **Frontend Requests to Backend**:
   * Implement a function in the Vue.js frontend to make requests to the Django API.
   * Send the uploaded document and user’s query to the backend using **Axios**.
   * Ensure that the frontend handles both successful and error responses appropriately.
2. **Handle Responses in Frontend**:
   * Once the backend returns the answer from the Gemini API, display it on the frontend in a user-friendly format.

**Milestone 6: Testing and Debugging**

**Steps:**

1. **Unit Testing**:
   * Write unit tests for both frontend and backend components.
   * Test document upload, query submission, and interaction with Google Gemini API.
2. **Integration Testing**:
   * Ensure that the complete flow from document upload to answer generation works smoothly.
   * Test edge cases, such as unsupported document formats or invalid queries.
3. **User Acceptance Testing (UAT)**:
   * Conduct testing with real users to identify usability issues.
   * Gather feedback and make any necessary adjustments.

**Milestone 7: Deployment**

**Steps:**

1. **Backend Deployment (Django)**:
   * Deploy the Django app on a cloud platform like **Heroku**, **DigitalOcean**, or **Google Cloud Platform**.
   * Ensure that the **Google Gemini API** credentials are securely managed.
2. **Frontend Deployment (Vue.js)**:
   * Build the production-ready Vue.js app.

bash

Copy code

npm run build

* + Deploy the static files on platforms like **Netlify**, **Vercel**, or **Firebase Hosting**.

1. **Ensure End-to-End Functionality**:
   * Ensure that both the frontend and backend are correctly connected in the production environment.
   * Test document upload and query processing end-to-end after deployment.

**Milestone 8: Maintenance and Future Enhancements**

**Steps:**

1. **Monitor Usage and Performance**:
   * Monitor the performance of the app in the live environment.
   * Ensure that the Google Gemini API usage stays within free tier limits or adjust the project as needed.
2. **Optimize and Update**:
   * Optimize the app for better performance (e.g., caching frequent queries, handling large documents).
   * Update the app based on user feedback and new features released by Google Gemini.
3. **Future Enhancements**:
   * Add support for additional document formats (e.g., images, tables).
   * Enhance the model’s ability to process long documents by breaking them into smaller chunks for the API.

**Conclusion**

This project involves developing a RAG app using **Vue.js** for the frontend, **Django** for the backend, and integrating **Google Gemini API** (Flash model) for generative responses based on document content. The key milestones include setting up the frontend and backend, integrating the Google Gemini API, and ensuring smooth communication between the two components. Testing, deployment, and maintenance are crucial for a successful launch and continued operation of the app.