Technical Description of a Next.js SaaS Project

Overview

This Next.js SaaS project allows users to upload PDF files and leverage OpenAI's GPT-3.5 language model to ask questions about the content of these files. The application includes authentication powered by Kinde Auth, a dashboard, individual file display pages, and a billing page. The key functionality revolves around parsing the PDF content, vectorizing it, and using OpenAI's capabilities to provide intelligent responses to user queries.

Key Features

1. **Authentication and Authorization**: User login and sign-up are managed using Kinde Auth.
2. **Dashboard**: A landing page that provides an overview of the user's account and uploaded files.
3. **File Upload and Processing**: Users can upload PDF files, which are then parsed and stored.
4. **Billing**: A billing page that handles subscription plans via Stripe.
5. **Question Answering**: Users can ask questions about their uploaded PDFs, and the system uses OpenAI to generate answers based on the document content.

Code Breakdown

Authentication Middleware

```
const middleware = async () => {
  const { getUser } = getKindeServerSession();
  const user = await getUser();

  if (!user || !user.id) {
    throw new Error("Unauthorized");
  }

  const subscriptionPlan = await getUserSubscriptionPlan();

  return { userId: user.id, subscriptionPlan };
};
```

This middleware function authenticates users using Kinde Auth and retrieves their subscription plan. If the user is not authenticated, an error is thrown.

File Upload Handling

```
const onUploadComplete = async ({ metadata, file }: { metadata: Awaited<ReturnType<type
  // Check if file already exists
  const existingFile = await db.file.findFirst({ where: { key: file.key } });
  if (existingFile) return;

  // Create new file record
  const createdFile = await db.file.create({
    data: { key: file.key, userId: metadata.userId, name: file.name, url: file.url, upl
  });

  try {
    // Fetch file and convert to blob
    const res = await fetch(file.url);
    const blob = await res.blob();
    const loader = new PDFLoader(blob);

    // Load and parse PDF content
    const pageLevelDocs = await loader.load();
    const pagesAmt = pageLevelDocs.length;

    // Check subscription plan limits
    const { subscriptionPlan } = metadata;
    const { isSubscribed } = subscriptionPlan;
    const isProExceeded = pagesAmt > PLANS.find((plan) => plan.name === "Pro")!.pagesPe
    const isFreeExceeded = pagesAmt > PLANS.find((plan) => plan.name === "Free")!.pages

    if ((isSubscribed && isProExceeded) || (!isSubscribed && isFreeExceeded)) {
      await db.file.update({ where: { id: createdFile.id }, data: { uploadStatus: "FAIL
      return;
    }

    // Vectorize and index the document
    const pineconeIndex = pinecone.Index("documentor");
    const embeddings = new OpenAIEmbeddings({ openAIApiKey: process.env.OPENAI_API_KEY
    await PineconeStore.fromDocuments(pageLevelDocs, embeddings, { pineconeIndex, names

    await db.file.update({ where: { id: createdFile.id }, data: { uploadStatus: "SUCCES
  } catch (err) {
    console.error(err);
    await db.file.update({ where: { id: createdFile.id }, data: { uploadStatus: "FAILED
  }
};
```

This function handles the complete lifecycle of a file upload:

1. It checks if the file already exists.
2. Creates a new record in the database.
3. Fetches and processes the PDF file.
4. Parses the PDF into pages.
5. Checks if the upload exceeds the subscription plan limits.
6. Vectorizes and indexes the document content using Pinecone and OpenAI embeddings.
7. Updates the upload status in the database.

File Router Configuration

```
export const ourFileRouter = {
  freePlanUploader: f({ pdf: { maxFileSize: "4MB" } })
    .middleware(middleware)
    .onUploadComplete(onUploadComplete),
  proPlanUploader: f({ pdf: { maxFileSize: "16MB" } })
    .middleware(middleware)
    .onUploadComplete(onUploadComplete),
} satisfies FileRouter;
```

Defines two file upload routes, one for free users and one for pro users, each with different file size limits and linked to the middleware and `onUploadComplete` functions.

Question Answering Endpoint

```
export const POST = async (req: NextRequest) => {
  const body = await req.json();
  const { getUser } = getKindeServerSession();
  const user = await getUser();
  const userId = user?.id;

  if (!userId) return new Response("Unauthorized", { status: 401 });

  const { fileId, message } = SendMessageValidator.parse(body);
  const file = await db.file.findFirst({ where: { id: fileId, userId } });
  if (!file) return new Response("File not found", { status: 404 });

  await db.message.create({ data: { fileId, userId, text: message, isUserMessage: true

  // Vectorize the message
  const embeddings = new OpenAIEmbeddings({ openAIApiKey: process.env.OPENAI_API_KEY })
  const pineconeIndex = pinecone.Index("documentor");
  const vectorStore = await PineconeStore.fromExistingIndex(embeddings, { pineconeIndex

  const results = await vectorStore.similaritySearch(message, 4);

  const prevMessages = await db.message.findMany({ where: { fileId }, orderBy: { update
  const formattedPrevMessages = prevMessages.map((m) => ({ role: m.isUserMessage ? "use

  const response = await openai.chat.completions.create({
    model: "gpt-3.5-turbo",
    temperature: 0,
    stream: true,
    messages: [
      { role: "system", content: "Use the following pieces of context (or previous conv
      { role: "user", content: `Use the following pieces of context (or previous conver
    ],
  });

  const stream = OpenAIStream(response, {
    async onCompletion(completion) {
      await db.message.create({ data: { fileId, userId, text: completion, isUserMessage
    },
  });
```

```
    return new StreamingTextResponse(stream);
  };
```

This endpoint handles the question-answering functionality:

1. Validates and authenticates the user.
2. Retrieves the relevant file and previous messages.
3. Vectorizes the user's question.
4. Searches for similar content within the indexed PDF.
5. Formats the context and previous messages for the OpenAI model.
6. Generates a response using OpenAI's chat completion.
7. Streams the response back to the user and logs it in the database.

Steps of Vectorization and Data Parsing

1. **File Upload and Initialization**:
   - When a user uploads a PDF file, the system first checks if the file already exists in the database. If not, it creates a new record and marks the file's upload status as "PROCESSING".
2. **Fetching and Converting the File**:
   - The system fetches the uploaded file from the provided URL.
   - It converts the fetched file into a binary large object (blob), making it easier to manipulate and process the file content.
3. **Parsing the PDF**:
   - Using a PDF parsing library (like `PDFLoader`), the system reads the PDF blob and extracts the textual content from each page of the document.
   - This step breaks down the PDF into manageable chunks (usually by page), allowing for more efficient processing and analysis.
4. **Checking Subscription Limits**:
   - The system checks the user's subscription plan to determine if the number of pages in the PDF exceeds the allowed limit for their plan. If it does, the upload status is marked as "FAILED".
5. **Vectorization**:
   - Vectorization involves converting textual content into numerical representations (vectors) that can be understood and processed by machine learning models.
   - The system uses OpenAI's embeddings, which are pre-trained models that convert text into high-dimensional vectors. These embeddings capture the semantic meaning of the text.
6. **Creating and Storing Vectors**:

- Each page or chunk of text from the PDF is passed through the embedding model to generate vectors.
- These vectors are stored in Pinecone, a vector database that allows for efficient similarity searches. Each vector is indexed with an identifier that links it to the original PDF file.

7. **Indexing in Pinecone**:
   - The vectors are stored in a specific namespace within Pinecone, ensuring that each document's vectors are kept separate.
   - Pinecone provides a scalable and efficient way to index and search through these vectors, allowing for quick retrieval based on similarity to other text.

Querying and Answer Generation

1. **User Question Input**:
   - When a user asks a question about an uploaded PDF, the system first logs the question in the database.
2. **Vectorizing the Question**:
   - The user's question is converted into a vector using the same embedding model used for the PDF content.
3. **Similarity Search**:
   - The vectorized question is used to perform a similarity search within the Pinecone index.
   - The search retrieves the most relevant vectors (chunks of text) from the indexed PDF that are similar to the user's question.
4. **Contextual Formatting**:
   - The system retrieves previous messages and formats them, along with the most relevant text chunks, into a coherent context for the OpenAI model.
5. **Generating the Response**:
   - OpenAI's language model uses the provided context and the user's question to generate a response.
   - The response is streamed back to the user in real-time and logged in the database for future reference.

Summary

The process of vectorization and data parsing in this Next.js SaaS project involves breaking down the PDF into text chunks, converting these chunks into vectors using OpenAI embeddings, and indexing them in Pinecone for efficient similarity searches. When a user asks a question, the system retrieves relevant text based on vector similarity and generates a response using OpenAI's language model. This ensures that the answers are contextually relevant to the content of the uploaded PDF.