Documentation for Knowledge Base Query System

Overview

This documentation provides an in-depth explanation of the key modules and logic within the Knowledge Base Query System. The code integrates GenKit AI capabilities to retrieve and generate responses based on an internal knowledge base, facilitating interactions between employees and the company's records for buyers, manufacturers, and departments such as merchandisers, QA managers, and logistics agents.

---

1. Knowledge Base Retriever

`kbRetriever`

Purpose: Serves as a reference to the local vector store for the knowledge base.

- **Defined as**: `devLocalRetrieverRef('knowledgeBase')`
- **Functionality**: Acts as the retriever for document queries within the company's knowledge base.
- **Integration**: Used by the `retrieve` function to fetch relevant documents based on input queries.

---

2. Knowledge Base QA Flow

`kbQAFlow`

Purpose: Defines the flow for querying the knowledge base and generating an AI-assisted response.

- **Parameters**:
    - `inputSchema`: Validates the input as a string.
    - `outputSchema`: Ensures the output is a string.

Logic:

1. **Document Retrieval**:
   - Queries the knowledge base using the `retrieve` function.
   - Fetches relevant documents matching the input query using the `kbRetriever`.
   - Restricts results to the top `k=1` most relevant document(s).
2. **Response Generation**:
   - Uses the `generate` function to create an AI response based on the retrieved documents and the employee's query.
   - Leverages `llama3.2` as the AI model with a tailored prompt.
3. **Prompt Details**:
   - Contextualizes the AI as a helpful assistant for the company's knowledge base.
   - Ensures the response adheres to:
     - Truthful answers based on provided documentation.
     - Asking clarifying questions when insufficient context is available.
     - Avoiding fabricated information.
4. **Output**:
   - The response text generated by the AI model is returned as the output of the flow.

Example Usage:

typescript
Copy code
```
const response = await kbQAFlow.execute("What are the supplier details
for QA managers?");
console.log(response);
```

---

3. Key Functions

3.1 `retrieve`

Purpose: Fetches documents related to the input query from the local vector store.

- **Parameters**:
  - `retriever`: Specifies the retriever (`kbRetriever`) to use for the query.
  - `query`: The input string to search within the knowledge base.
  - `options`: Additional configurations (e.g., `k: 1` limits results to the top match).
- **Returns**: A list of relevant documents.

Example Usage:

typescript
Copy code
```
const docs = await retrieve({
  retriever: kbRetriever,
  query: "Supplier contact for logistics agents",
  options: { k: 1 },
});
```

3.2 `generate`

Purpose: Produces an AI-generated response based on the retrieved documents and input query.

- **Parameters**:
  - `model`: Specifies the AI model to use (`ollama/llama3.2`).
  - `prompt`: A detailed instruction for the AI to follow while generating the response.
  - `context`: The content retrieved from the knowledge base.
- **Returns**: The generated response as a string.

Example Usage:

typescript
Copy code
```
const llmResponse = await generate({
  model: 'ollama/llama3.2',
  prompt: "Provide supplier details based on the following query: 'QA
manager responsibilities'",
  context: docs,
});
```

---

4. Design Principles Used

4.1 Error Handling

- Errors are not explicitly handled in the current code. Incorporating `try/catch` blocks can improve robustness by logging or handling retrieval or generation failures.

4.2 Modular Design

- The functions (`retrieve` and `generate`) are decoupled from the flow logic, enabling reuse and testability.

4.3 Scalability

- The use of `devLocalRetrieverRef` and AI models like `llama3.2` ensures scalability to larger datasets or more complex queries.

4.4 User-Centric Design

- Responses are tailored to the context and ensure clarity by asking for more information if the query lacks sufficient details.

---

5. Conclusion

The Knowledge Base Query System integrates seamlessly with GenKit AI and local vector stores to provide a robust, modular solution for querying and generating insights from the company's knowledge base. This system ensures accurate, context-sensitive responses for employees across various departments, enhancing efficiency and productivity in accessing critical records.

Important Notes:

Indexer Flows: The index is responsible for keeping track of your documents in such a way that you can quickly retrieve relevant documents given a specific query. This is most often accomplished using a vector database, which indexes your documents using multidimensional vectors called embeddings. A text embedding (opaquely) represents the concepts expressed by a passage of text; these are generated using special-purpose ML models. By indexing text using its embedding, a vector database is able to cluster conceptually related text and retrieve documents related to a novel string of text (the query).

Before you can retrieve documents for the purpose of generation, you need to ingest them into your document index. A typical ingestion flow does the following:

Split up large documents into smaller documents so that only relevant portions are used to augment your prompts – "chunking". This is necessary because many LLMs have a limited context window, making it impractical to include entire documents with a prompt.

Genkit doesn't provide built-in chunking libraries; however, there are open source libraries available that are compatible with Genkit.

1. Generate embeddings for each chunk. Depending on the database you're using, you might explicitly do this with an embedding generation model, or you might use the embedding generator provided by the database.
2. Add the text chunk and its index to the database.
3. You might run your ingestion flow infrequently or only once if you are working with a stable source of data. On the other hand, if you are working with data that frequently changes, you might continuously run the ingestion flow (for example, in a Cloud Firestore trigger, whenever a document is updated).
4. Embedders - An embedder is a function that takes content (text, images, audio, etc.) and creates a numeric vector that encodes the semantic meaning of the original content.

Retrievers: A retriever is a concept that encapsulates logic related to any kind of document retrieval. The most popular retrieval cases typically include retrieval from vector stores.

Vector DB (Cloud Firestore) used. It indexes documents and data and stores them in multidimensional vectors. Vector stores are used to store embeddings, which are the numeric vectors that encode the semantic meaning of the original content.

RAG (retrieval-augmented generation) architecture implemented through indexer, embedder, retriever, and generator. First, we use an indexer and index the documents, storing text embeddings in a local vector store. Then, to generate a response, a retriever is initialized to get relevant documents. The LLM will use the text embeddings retrieved to generate a relevant response.

Indexer flow can be paired with a Firestore trigger to run with update embeddings when data/documents are uploaded.

Text embedder was Vertex AI with TextEmbeddingGecko.

Retriever used with Llama 3.2 to generate response with tuned prompting to give a more relevant response.

API route is called by handleChat function in aiButton.

Documentation for AI Button Component

Overview

This document details the functionality, logic, and design principles of the `AiButton` React component. The component serves as a user interface element that provides two primary functionalities:

1. **Summarizing selected PDF files**
2. **Interacting with an AI chatbot.**

The component leverages various libraries, including `pdfjs-dist` for PDF parsing and `ReactMarkdown` for rendering markdown content, ensuring a seamless and interactive experience for users.

---

1. Component Functionality

Primary Features

1. **Toggleable Interface**:
   - A circular button at the bottom-right toggles the visibility of an interaction card.
2. **Modes**:
   - **Summarize Mode**: Allows users to select a PDF file and retrieve a summarized text.
   - **Chat Mode**: Enables users to chat with an AI assistant.
3. **Input Handling**:
   - Dynamic input handling based on the selected mode (file selection for summarizing or text input for chatting).
4. **Keyboard Accessibility**:
   - Uses the `Escape` key to close the card and the `Enter` key for executing mode-specific actions.

---

2. Key State Variables

2.1 `isCardVisible`

Purpose: Tracks the visibility state of the card.

- **Default**: `false`
- **Actions**: Toggled by the circular button.

2.2 `mode`

Purpose: Determines the active functionality (`"summarize"` or `"chat"`).

- **Default**: `"summarize"`
- **Actions**: Changed using tab buttons in the card.

2.3 `inputValue`

Purpose: Stores user input in chat mode.

2.4 `fileSelectedForSummary`

Purpose: Tracks the selected file for summarization.

- **Type**: `SummarySearchResult | null`

2.5 `summaryContent`

Purpose: Stores the summarized content of the selected file.

2.6 `chatHistory`

Purpose: Maintains the conversation history between the user and the AI assistant.

2.7 `loading` / `chatLoading`

Purpose: Indicates ongoing API calls for summarization and chat functionalities.

---

3. Component Logic

3.1 PDF Summarization

Process:

1. User selects a PDF file using the `SearchBarAI` component.

2. `handleSummarizeClick` fetches the file, extracts its text content using `pdfjs-dist`, and sends it to the server for summarization.
3. The server processes the text and returns a summarized version, which is displayed in the card.

Error Handling:

- Alerts the user if no file is selected.
- Displays a fallback error message if PDF processing or summarization fails.

3.2 Chat Functionality

Process:

1. User enters a query into the chat input field.
2. `handleChat` sends the query to the `/api/chatbot` endpoint and appends the user's query and the AI's response to `chatHistory`.
3. The response is rendered as a conversation.

Error Handling:

- Logs errors to the console and displays a fallback message if the API call fails.

3.3 Keyboard Accessibility

- **Escape**: Closes the card.
- **Enter**: Executes the current mode's primary action:
  - Summarizes the selected file in Summarize mode.
  - Sends the chat query in Chat mode.

---

4. UI Components

4.1 Circle Button

Purpose: Toggles the visibility of the card. **Class**: `styles.circleButton`

4.2 Tab Buttons

Purpose: Switches between Summarize and Chat modes. **Classes**:

- Active tab: `styles.active`
- Summarize tab: `#summarizeTab`

- Chat tab: `#chatTab`

4.3 Content Display Area

- **Summarize Mode**: Shows the `summaryContent` with a spinner while loading.
- **Chat Mode**: Displays `chatHistory` with user and bot messages styled differently.

4.4 Input Area

- **Summarize Mode**:
  - Includes the `SearchBarAI` component for file selection.
  - An action button triggers summarization.
- **Chat Mode**:
  - Includes a text input for user queries and a button to send queries.

---

5. Design Principles

5.1 Modular Design

- The `SearchBarAI` component and API logic are decoupled, enabling reuse and easier testing.

5.2 Accessibility

- Implements keyboard shortcuts for toggling the card and executing actions.

5.3 User Feedback

- Loading spinners and error messages ensure clear user communication during asynchronous operations.

---

6. Example Usage

jsx

Copy code

```jsx
import AiButton from "./AiButton";
import { SummarySearchResult } from "../types";


const paths: SummarySearchResult[] = [

  { name: "Sample.pdf", downloadURL: "/path/to/sample.pdf", author:
"Author", uploadDate: "2024-11-27" },

];


function App() {

  return <AiButton paths={paths} />;

}

export default App;
```

---

7. API Endpoints

`/api/summarize`

Purpose: Processes and summarizes text content.

- **Method**: POST

Body:
json
Copy code

```
{

  "text": "Extracted text from the PDF",

  "metadata": "File metadata (title, author, upload date)"

}
```

-

Response:
json
Copy code
```
{

  "summary": "Summarized content"

}
```

-

`/api/chatbot`

Purpose: Provides AI-generated responses based on user queries.

- **Method**: POST

Body:
json
Copy code
```
{

  "message": "User's chat query"

}
```

-

Response:
json
Copy code
```
{

  "response": "AI-generated response"

}
```

---

8. Conclusion

The `AiButton` component combines robust PDF summarization and chatbot functionalities into a user-friendly interface. Its modular design, keyboard accessibility, and clear user feedback mechanisms make it an essential tool for enhancing productivity and engagement.

Documentation for User Permissions

Overview

The Permissions functionality manages access control for various departments and features within the application. It dynamically evaluates whether a user has sufficient permissions to view or interact with specific sections based on their role or department association. This system ensures that only authorized users can access or modify department-specific content, enhancing security and usability.

---

1. Features

Department Mapping

- **Purpose**: Maps department keys to Firestore document IDs for validating access.

Implementation:
javascript
Copy code
```
const departmentMapping = {
    qa: 'Eq2IDInbEQB5nI5Ar6Vj',
    hr: 'NpaV1QtwGZ2MDNOGAlXa',
    logistics: 'KZm56fUOuTobsTRCfknJ',
    merchandising: 'ti7yNByDOzarVXoujOog',
};
```

-

Admin Check

- **Purpose**: Verifies whether a signed-in employee is an admin.
- **Process**:
    - Retrieves admin employee references from the Firestore `Company` document.
    - Fetches and compares the signed-in employee's name against the admin list.
    - Updates the `isAdmin` state to reflect the employee's admin status.

Key Functionality:
javascript
Copy code
```
const fetchAdmins = async () => {
    const companyDocRef = doc(db, 'Company', 'mh3VZ5IrZjubXUCZL381');
    const companyData = (await getDoc(companyDocRef)).data();
    const admins = companyData?.admins || [];
    const adminNames = await Promise.all(admins.map(ref =>
getDoc(ref).then(snap => snap.data()?.name)));
    setIsAdmin(adminNames.includes(employeeProfile?.name));
};
```

-

Department Access Control

- **Purpose**: Determines if the signed-in user has access to a department.

Implementation:
javascript
Copy code
```
const isDepartmentEnabled = (departmentKey) => {
    const departmentId = departmentMapping[departmentKey];
    return userDepartments.includes(departmentId);
};
```

-
- **Example Use**:
    - Grants or restricts access to the **Quality Assurance** department based on the user's permissions.

javascript
Copy code
```
{isDepartmentEnabled('qa') || isAdmin ? (
    <Link href="/departments/qa">
        <div className={styles.topButtons} style={{ opacity: 1 }}>
            Quality Assurance
        </div>
    </Link>
) : (
    <div className={styles.topButtons} style={{ opacity: 0.5, cursor:
'not-allowed' }} aria-disabled="true">
        Quality Assurance
    </div>
)}
```

-

---

2. UI Components

Department Links

- **Purpose**: Dynamic links to department pages, enabled or disabled based on access permissions.
- **Behavior**:
  - Active links are fully styled and clickable.
  - Disabled links have reduced opacity and a "not-allowed" cursor with `aria-disabled="true"` for accessibility.

Create Employee Button

- **Purpose**: Restricts access to the employee creation feature to admin users.
- **Behavior**:
  - Fully enabled for admins.
  - Disabled and styled accordingly for non-admin users.

---

3. Design Principles

1. Role-Based Access Control

- Dynamically determines access using the `isAdmin` state and `isDepartmentEnabled` function.

2. User Feedback

- Clear visual cues for restricted sections (reduced opacity and "not-allowed" cursor).
- Ensures accessibility by using the `aria-disabled` attribute for disabled buttons.

3. Security

- Permissions are validated dynamically through Firestore data and mappings, ensuring unauthorized users cannot access restricted resources.

---

4. Example Usage

Displaying the Quality Assurance Department Link:

javascript

Copy code
```
{isDepartmentEnabled('qa') || isAdmin ? (
    <Link href="/departments/qa">
        <div style={{ opacity: 1 }}>Quality Assurance</div>
    </Link>
) : (
    <div style={{ opacity: 0.5, cursor: 'not-allowed' }}
aria-disabled="true">
        Quality Assurance
    </div>
)}
```

---

5. Key States and Variables

Variable	Purpose	Default Value
`departmentMapping`	Maps department keys to Firestore document IDs.	N/A
`isAdmin`	Tracks whether the signed-in employee is an admin.	`false`
`userDepartments`	List of department IDs the user has access to.	`[]`
`isDepartmentEnabled`	Function to check if a department is accessible by the user.	N/A

---

6. Conclusion

The Permissions functionality integrates seamlessly with the UI, providing a robust system for managing department-specific and role-based access. It ensures secure, intuitive, and accessible navigation while maintaining flexibility for future expansion.