API Documentation

This document provides detailed information about the functions and modules available in the User Reading Analysis and Book Recommendation System.

Table of Contents

Data Source Module

The Data Source module (`src/libs/data_source.ts`) provides functions for interacting with the Snowflake database to retrieve user reading data and book information. It implements a connection pool pattern with a single, reusable database connection.

`getDbConnection()`

Establishes and returns a connection to the Snowflake database. Uses the singleton pattern - creates a new connection if one doesn't exist, otherwise returns the existing connection.

Returns:

- `Promise<Connection>`: A Promise that resolves to a Snowflake connection

Example:

```
// Get a database connection
const connection = await getDbConnection();

// Use the connection to execute a query
connection.execute({
  sqlText: "SELECT * FROM my_table LIMIT 10",
  complete: (err, stmt, rows) => {
    if (err) {
      console.error('Error executing query:', err);
    } else {
      console.log('Query results:', rows);
    }
  }
});
```

`getActiveUsers(days, minActivityCount)`

Retrieves a list of active users based on their reading activity. This function queries the database for users who have been active readers within a specified timeframe.

Parameters:

- `days` (number, optional): Number of days to look back for activity (default: 14)
- `minActivityCount` (number, optional): Minimum number of reading sessions required (default: 5)

Returns:

- `Promise<string[]>`: A Promise that resolves to an array of user IDs

Example:

```
// Get users who have read at least 5 books in the last 7 days
const activeUsers = await getActiveUsers(7, 5);
console.log(`Found ${activeUsers.length} active users`);

// Process each active user
for (const userId of activeUsers) {
  // Do something with each user ID
  console.log(`Processing user: ${userId}`);
}
```

`getUserReadBooks(userId, limit)`

Retrieves books that a specific user has read. The books are returned in reverse chronological order (most recent first).

Parameters:

- `userId` (string): The ID of the user
- `limit` (number, optional): Maximum number of books to retrieve (default: 5)

Returns:

- `Promise<any[]>`: A Promise that resolves to an array of book objects with the following properties:
  - `event_time`: When the book was read
  - `title`: Title of the book
  - `author`: Author of the book
  - `isbn`: ISBN of the book
  - `language_code`: Language code of the book
  - `genre`: Genre of the book
  - `publisher`: Publisher of the book
  - `word_count`: Word count of the book
  - `default_categories`: Categories of the book
  - `book_id`: Unique identifier of the book

Example:

```
// Get the 10 most recent books read by a user
const userId = 'user123';
const recentBooks = await getUserReadBooks(userId, 10);

// Display the titles of the books
recentBooks.forEach((book, index) => {
  console.log(`${index + 1}. ${book.title} (Read on:
${book.event_time})`);
});
```

`getBookInfo(bookId)`

Retrieves detailed information about a specific book. It's used to get more comprehensive information about a book when building the reading history for recommendation processing.

Parameters:

- `bookId` (string): The permanent ID of the book

Returns:

- `Promise<string>`: A Promise that resolves to the book's description

Example:

```
// Get detailed information for a specific book
try {
  const bookId = '12345-1';
  const bookDescription = await getBookInfo(bookId);
  console.log(`Book Description: ${bookDescription.substring(0,
100)}...`);
} catch (error) {
  console.error(`Failed to get book info: ${error}`);
}
```

`fetchAllProductionBooks()`

Fetches all books from the production database and saves them to a temporary file. This function is used to prepare data for the OpenAI Assistant.

Process:

1. Queries the database for all books
2. Formats each book as a JSON object
3. Writes the data to a temporary file with a unique name
4. Returns the path to this file

Returns:

- `Promise<string>`: A Promise that resolves to the path of the temporary file

Example:

```
// Fetch all books and save to a temporary file
try {
  const tempFilePath = await fetchAllProductionBooks();
  console.log(`Books data saved to: ${tempFilePath}`);

  // Use the file for OpenAI Assistant creation
  const assistantId = await ensureAssistant(tempFilePath);

  // Clean up when done
  fs.unlinkSync(tempFilePath);
} catch (error) {
  console.error(`Failed to fetch books: ${error}`);
}
```

OpenAI Assistant Module

The OpenAI Assistant module (`src/libs/openai_assistant.ts`) provides functions for interacting with OpenAI's Assistant API to analyze user reading habits and generate book recommendations. It also includes functionality for tracking token usage and calculating costs.

`createVectorStoreWithFile(libraryDataPath)`

Creates a vector store and uploads a library data file to it.

Process:

1. Creates a new vector store in OpenAI
2. Uploads the library data file to OpenAI
3. Links the uploaded file to the vector store for search capabilities

Parameters:

- `libraryDataPath` (string): Path to the library data file

Returns:

- `Promise<string>`: A Promise that resolves to the vector store ID

Example:

```
// Create a vector store with book data
const libraryDataPath = './data/library-catalog.json';
const vectorStoreId = await createVectorStoreWithFile(libraryDataPath);
console.log(`Vector store created: ${vectorStoreId}`);
```

`ensureAssistant(libraryDataPath)`

Creates an OpenAI Assistant with file search capabilities and function tools.

Process:

1. Creates a vector store with the library data
2. Sets up an Assistant with specific instructions
3. Configures tools for file search and book recommendations

Parameters:

- `libraryDataPath` (string): Path to the library data file

Returns:

- `Promise<string>`: A Promise that resolves to the assistant ID

Example:

```
// Create an assistant with book recommendation capabilities
const assistantId = await ensureAssistant('./data/books.json');
console.log(`Assistant created: ${assistantId}`);
```

`analyzeUserInterest(assistantId, userData, userId)`

Analyzes a user's reading interests based on their reading history and tracks token usage.

Process:

1. Creates a new thread for the conversation
2. Sends the user's reading history to the assistant
3. Extracts a summary of the user's reading interests
4. Tracks token usage for this operation

Parameters:

- `assistantId` (string): The ID of the OpenAI Assistant
- `userData` (string): The user's reading history data
- `userId` (string): The user ID for token tracking

Returns:

- `Promise<string>`: A Promise that resolves to a summary of the user's interests

Example:

```
// Analyze a user's reading interests
const userData = "Recent books: Book 1 (Fantasy), Book 2 (Science)...";
const interests = await analyzeUserInterest(assistantId, userData,
"user123");
```

```
console.log(`User interests: ${interests}`);
// Output might be: "science fiction and educational content"
```

searchBooksByInterest(assistantId, userData, userId)

Searches for books that match a user's interests and tracks token usage.

Process:

1. Creates a new thread for the conversation
2. Sends the user's reading history to the assistant
3. Instructs the assistant to use file search to find relevant books
4. Returns structured book recommendations with reasons
5. Tracks token usage for this operation

Parameters:

- `assistantId` (string): The ID of the OpenAI Assistant
- `userData` (string): The user's reading history data
- `userId` (string): The user ID for token tracking

Returns:

- `Promise<any[]>`: A Promise that resolves to an array of recommended books with the following properties:
  - `book_id`: Unique identifier of the book
  - `book_title`: Title of the book
  - `reason`: Reason for recommendation

Example:

```
// Get book recommendations for a user
const recommendations = await searchBooksByInterest(
  assistantId,
  userReadingHistory,
  "user123"
);

// Display recommendations
recommendations.forEach(book => {
  console.log(`Book: ${book.book_title}`);
  console.log(`Reason: ${book.reason}`);
});
```

monitorRun(runId, threadId, isInterestAnalysis, userId)

Monitors an OpenAI Assistant run, processes the results, and tracks token usage.

Process:

1. Polls the run status until completion or an action is required
2. Handles function calls from the assistant
3. Processes book recommendations
4. Tracks token usage for cost calculation
5. Returns the appropriate results based on the operation type

Parameters:

- `runId` (string): The ID of the run
- `threadId` (string): The ID of the thread
- `isInterestAnalysis` (boolean, optional): Whether this is an interest analysis run (default: false)
- `userId` (string): The user ID for token tracking

Returns:

- `Promise<any>`: A Promise that resolves to the run results (string for interest analysis, array of book objects for recommendations)

Example:

```
// For interest analysis
const analysisResult = await monitorRun(run.id, thread.id, true,
"user123");
// Returns a string like "science fiction and educational content"

// For book recommendations
const recommendations = await monitorRun(run.id, thread.id, false,
"user123");
// Returns an array of book objects with IDs, titles, and reasons
```

`deleteAssistant(assistantId)`

Deletes an OpenAI Assistant and its associated resources.

Process:

1. Retrieving the assistant details
2. Finding all associated vector stores
3. Deleting all files from each vector store
4. Deleting each vector store
5. Deleting the assistant itself

Parameters:

- `assistantId` (string): The ID of the assistant to delete

Example:

```
// Clean up all resources when done
await deleteAssistant(assistantId);
console.log("All resources cleaned up successfully");
```

tokenTracker

A singleton instance of the TokenTracker class for tracking OpenAI API token usage and calculating costs.

Example:

```
// Get total cost for all operations
const totalCost = tokenTracker.getTotalCost();
console.log(`Total API cost: $${totalCost.toFixed(6)}`);

// Print a detailed usage and cost summary
tokenTracker.printSummary();
```

Prompt Templates Module

The Prompt Templates module (`src/libs/prompt_templates.ts`) provides templates for OpenAI prompts and formatting. These templates define:

- Instructions for the Assistant
- Formats for user reading history
- Analysis and recommendation prompts
- Function descriptions and output formats

Constants

OPENAI_ASSISTANT_INSTRUCTION

Core instructions for the OpenAI Assistant. This defines the assistant's primary role and capabilities, instructing it to analyze user reading history to identify interests, use the file_search tool to find relevant books, and return recommendations with specific fields.

OPENAI_USER_INTEREST_ANALYSIS_PROMPT

Prompt template for analyzing user interests based on reading history. This prompt asks the AI to identify patterns in a user's reading history and determine their interests and preferences.

OPENAI_USER_RECOOMMANDATION_PROMPT

Prompt template for recommending books based on user reading history. This prompt specifically instructs the AI to use the file_search functionality to find and recommend books from the vector store.

OPENAI_USER_READING_HISTORY_RECORD

Template for formatting a single reading history record. This template is used to format each book in a user's reading history in a consistent way.

`OPENAI_ANALYSIS_FUNCTION_DESCRIPTION`

Description for the recommendation function. This text is used in the function tool definition for the OpenAI Assistant. It instructs the AI on how to analyze user reading history and format the results for the recommend_books function.

`OPENAI_ANALYSIS_FUNCTION_RESULT_DESCRIPTION`

Description for the recommendation function result. This text defines the expected format of the recommendation_summary field in the function output.

`DEBUG_OUTPUT_TEMPLATE`

Template for formatting debug output. This template creates a standardized console output format for displaying user analysis results during the recommendation process.

Example:

```
import { OPENAI_USER_READING_HISTORY_RECORD } from
'./libs/prompt_templates';

// Format a reading history record
const formattedRecord = OPENAI_USER_READING_HISTORY_RECORD
  .replace("{event_time}", "2023-01-15")
  .replace("{book_title}", "The Great Gatsby")
  .replace("{book_desc}", "A novel about the American Dream");
```

Token Tracking Module

The Token Tracking module (`src/libs/token_tracker.ts`) provides functionality for tracking OpenAI API token usage and calculating associated costs. It helps monitor the financial impact of using OpenAI's API services by tracking token usage per user and per operation.

Interfaces and Types

`TokenUsage`

Represents token usage data for a single API call.

Properties:

- `prompt_tokens` (number): Number of tokens in the input/prompt
- `completion_tokens` (number): Number of tokens in the output/completion
- `total_tokens` (number): Total tokens (prompt + completion)
- `operation` (string): Description of the operation (e.g., "Interest Analysis", "Book Recommendations")

- `cached?` (boolean, optional): Whether cached tokens were used for input (affects pricing)

`UserCostData`

Represents cost data for a single user.

Properties:

- `userId` (string): Unique identifier for the user
- `usages` (TokenUsage[]): Array of individual token usage records for this user
- `totalPromptTokens` (number): Total number of standard (non-cached) input tokens used
- `totalCachedPromptTokens` (number): Total number of cached input tokens used
- `totalCompletionTokens` (number): Total number of output tokens generated
- `totalTokens` (number): Total tokens (prompt + completion)
- `cost` (number): Total cost in USD for all operations

Class: `TokenTracker`

`constructor(model)`

Initialize a new TokenTracker with the specified model.

Parameters:

- `model` (string, optional): The OpenAI model being used (defaults to 'gpt-4o')

Example:

```
// Create with default model (gpt-4o)
const tracker = new TokenTracker();

// Create with a specific model
const tracker = new TokenTracker('gpt-3.5-turbo');
```

`addUsage(userId, usage)`

Add token usage data for a user. This method records token usage for a specific operation and calculates the associated cost based on current pricing.

Parameters:

- `userId` (string): The user ID
- `usage` (TokenUsage): The token usage data

Example:

```
tokenTracker.addUsage('user123', {
  prompt_tokens: 1500,
  completion_tokens: 300,
```

```
  total_tokens: 1800,
  operation: 'Book Recommendations',
  cached: false
});
```

`getUserCost(userId)`

Get cost data for a specific user.

Parameters:

- `userId` (string): The user ID

Returns:

- `UserCostData | undefined`: The user's cost data or undefined if user not found

Example:

```
const userCost = tokenTracker.getUserCost('user123');
if (userCost) {
  console.log(`User ${userCost.userId} spent
$${userCost.cost.toFixed(6)}`);
  console.log(`Total tokens: ${userCost.totalTokens}`);
}
```

`getAllUserCosts()`

Get cost data for all users.

Returns:

- `UserCostData[]`: Array of all users' cost data

Example:

```
const allUserCosts = tokenTracker.getAllUserCosts();
allUserCosts.forEach(userData => {
  console.log(`User ${userData.userId}: $${userData.cost.toFixed(6)}`);
});
```

`getTotalCost()`

Get total cost across all users.

Returns:

- `number`: Total cost in USD

Example:

```
const totalCost = tokenTracker.getTotalCost();
console.log(`Total API cost: $${totalCost.toFixed(6)}`);
```

`getTotalUsage()`

Get total token usage across all users.

Returns:

- Object containing total token counts:
  - `prompt_tokens` (number): Total input tokens
  - `cached_prompt_tokens` (number): Total cached input tokens
  - `completion_tokens` (number): Total output tokens
  - `total_tokens` (number): Sum of all tokens

Example:

```
const usage = tokenTracker.getTotalUsage();
console.log(`Total tokens used: ${usage.total_tokens.toLocaleString()}`);
console.log(`Input tokens: ${usage.prompt_tokens.toLocaleString()}`);
console.log(`Output tokens: ${usage.completion_tokens.toLocaleString()}`);
```

`printSummary()`

Print a formatted summary of token usage and costs.

Example:

```
// After tracking usage for multiple operations
tokenTracker.printSummary();
```

Output example:

```
💰 Token Usage and Cost Summary 💰
==================================

User ID: user123
  Total Tokens: 5,234
    - Input Tokens: 4,850
    - Output Tokens: 384
  Total Cost: $0.022158
  Operation Breakdown:
    - Interest Analysis: 1,245 tokens ($0.004500)
```

```
    - Book Recommendations: 3,989 tokens ($0.017658)

📊 TOTAL SUMMARY
  Total Input Tokens: 4,850
  Total Output Tokens: 384
  Total Tokens: 5,234
  TOTAL COST: $0.022158
```

Exported Instance

`tokenTracker`

A singleton instance of the TokenTracker class configured with the model specified in the environment variables or defaulting to 'gpt-4o'.

OpenAI Pricing Configuration

The OpenAI Pricing Configuration module (`src/config/openai_pricing.ts`) provides centralized configuration for OpenAI API pricing rates. It allows for easy updates when OpenAI changes their pricing structure.

Interfaces and Types

`ModelPricing`

Interface defining the pricing structure for an OpenAI model.

Properties:

- `input` (number): Cost per input token (in $)
- `cached_input` (number): Cost per cached input token (in $) for repeated requests
- `output` (number): Cost per output token (in $)

`OpenAIModel`

Type defining the supported OpenAI models in this application: 'gpt-4o' | 'gpt-4' | 'gpt-3.5-turbo'

Constants

`OPENAI_PRICING`

Pricing configuration for various OpenAI models. Values are expressed as cost per token ($/token).

Example:

```
import { OPENAI_PRICING } from '../config/openai_pricing';

// Get pricing for GPT-4o
const model = 'gpt-4o';
const pricing = OPENAI_PRICING[model];
```

```
// Calculate cost for 1000 input tokens and 500 output tokens
const inputCost = 1000 * pricing.input;
const outputCost = 500 * pricing.output;
const totalCost = inputCost + outputCost;

console.log(`Cost for API call: $${totalCost.toFixed(6)}`);
```

Main Application

The main application (`src/run_task.ts`) orchestrates the book recommendation process:

1. Fetches book data and creates a temporary file
2. Creates an OpenAI Assistant with the book data
3. Processes active users and analyzes their reading history
4. Generates and displays book recommendations
5. Tracks token usage and calculates costs
6. Cleans up resources and displays a cost summary

Example:

```
// Run the application with default settings (process 1 user)
npm start

// Run the application and process 5 users
npm start -- 5
```

Output includes:

- User reading interests analysis
- Personalized book recommendations with URLs and reasons
- Token usage and cost breakdown for each user
- Total cost summary for the entire run