**Updated Blueprint for Spotlight Project APIs and System Architecture**

**1. Objective**

The API framework aims to deliver a scalable, efficient, and dynamic solution for managing enriched media metadata to power niche-specific applications. Each app is built as a configurable template, with content determined by a flexible **keyword library**. The prototype focuses on AIK football, but the solution is designed to be reusable across various topics, ensuring no hardcoding of keywords or specific references within the codebase.

**2. Key Design Principles**

        1.        **No Hardcoding of Keywords**:

        •        The **keyword library** serves as the single source of truth for keyword definitions and variations.

        •        The library can be dynamically updated to support new topics, ensuring adaptability across markets without modifying the underlying codebase.

        2.        **Standardized Tools and Technologies**:

        •        **Firebase and Firestore**: Used for data storage and processing, ensuring reliable and scalable infrastructure.

        •        **GCP Security and Authorization**: The GCP security framework will handle authentication and access control, ensuring compliance and robust user data protection.

        •        **Custom Frontend Interface**: A custom-built frontend hosted on Firebase will serve as the primary user interface for apps, ensuring seamless integration with backend services.

        •        **Standard Media APIs**: Includes YouTube API and similar services to fetch data in compliance with platform guidelines.

        •        **GitHub for Version Control**: Used for managing the codebase, revisions, and deployment with GitHub Copilot as the main developer assistant.

        •        **OpenAI**: Used for metadata enrichment and internal AI-driven processing of media data.

        3.        **Centralized External API Calls**:

        •        All calls to external APIs (e.g., YouTube API) are managed centrally by the admin or scheduled processes, avoiding unnecessary costs and complexity.

        •        External calls fetch as much data as possible (e.g., full descriptions, transcripts) to minimize redundant requests.

**3. Core Workflow**

        1.        **Admin/Backend Workflow**:

        •        **API Query Management**:

        •        Admin inputs filters for queries, including options like usePageToken to fetch results exceeding 50 documents.

        •        Queries fetch enriched data (e.g., keywords, descriptions, transcripts) from external APIs.

        •        **AI-Driven Data Processing**:

        •        Retrieved media metadata is analyzed and processed by AI to add value such as:

        •        **Keyword Timestamps**: Links to specific segments or chapters of the media content.

        •        **Multi-Document Keys**: Variations of documents saved with unique keys (e.g., videoID\_keyword\_keywordtimestamp).

        2.        **End-User Workflow**:

        •        **Custom Frontend Interface**:

        •        Hosted on Firebase, this interface will display enriched metadata in a user-friendly feed.

        •        Users can fetch data in batches of 10 documents per scroll, ensuring a smooth and engaging experience.

        •        **Scroll-Based Fetching**:

        •        Users interact only with the **local database**, requesting metadata in small batches.

        •        **Feed Display**:

        •        Enriched metadata is displayed, with direct links to specific media segments.

        3.        **Defect Management**:

        •        Open bugs and filter-related issues are tracked in a defect list for resolution.

        •        Example: Some filter options in the YouTube API are currently not working as intended.

**4. Architectural Highlights**

        1.        **Flexibility and Reusability**:

        •        The solution is a fully configurable template adaptable to any topic or niche by updating the keyword library.

        •        Designed to scale across multiple apps with no topic-specific hardcoding.

        2.        **AI Integration**:

        •        OpenAI powers internal processing for metadata enrichment, creating added value in the database.

        3.        **Data Pipeline**:

        •        **External Fetching**:

        •        Fetch metadata from YouTube API and similar sources based on admin-defined filters.

        •        **Local Processing**:

        •        AI processes fetched data to generate enriched metadata with additional layers of information.

        •        **Staging and Storage**:

        •        Store enriched metadata in Firebase Firestore for fast retrieval by end-user apps.

        4.        **Custom Frontend**:

        •        The custom-built interface will:

        •        Showcase enriched data feeds.

        •        Offer seamless scrolling and fetching.

        •        Be fully hosted on Firebase for optimized performance.

        5.        **GCP Security and Authorization**:

        •        Implement GCP’s security frameworks to:

        •        Authenticate users securely.

        •        Protect user data and comply with GDPR requirements.

        •        Leverage GCP’s role-based access control (RBAC) for sensitive admin tasks.

**5. Operational Policies**

        1.        **Cost Management**:

        •        APIs (e.g., OpenAI, YouTube) are configured to minimize unnecessary usage.

        •        APIs must never autonomously switch to higher-cost models without explicit approval from Mattias, including a clear justification.

        2.        **Security and Compliance**:

        •        APIs interact only with approved services to ensure data privacy and integrity.

        •        No direct API requests from end-user apps to external services.

**6. Next Steps**

        1.        **Setting Up API Roles**:

        •        Define roles like “Markus” for architectural oversight and “Ulrich” for AI-driven metadata enrichment.

        •        Add a role for “Tomiko” to oversee frontend development.

        2.        **Custom Frontend Development**:

        •        Design and build the user interface, focusing on usability and scalability.

        3.        **GCP Security Setup**:

        •        Configure GCP authentication and authorization for secure access control.

        4.        **Defect Resolution**:

        •        Address open bugs, especially related to YouTube API filters, before scaling.

        5.        **Testing and Iteration**:

        •        Test the system end-to-end (E2E) with real user interactions, ensuring smooth data flow and seamless user experience.

This updated blueprint integrates the custom frontend and GCP security concept, ensuring the prototype aligns with long-term scalability and operational goals.

Updated Blueprint for Spotlight Project APIs and System Architecture

**1. Objective**

The API framework aims to deliver a scalable, efficient, and dynamic solution for managing enriched media metadata to power niche-specific applications. Each app is built as a configurable template, with content determined by a flexible keyword library. The prototype focuses on AIK football, but the solution is designed to be reusable across various topics, ensuring no hardcoding of keywords or specific references within the codebase.

**2. Key Design Principles**

* 1. No Hardcoding of Keywords:
  2. The keyword library serves as the single source of truth for keyword definitions and variations.
  3. The library can be dynamically updated to support new topics, ensuring adaptability across markets without modifying the underlying codebase.
  4. Standardized Tools and Technologies:
  5. Firebase and Firestore: Used for data storage and processing, ensuring reliable and scalable infrastructure.
  6. GCP Security and Authorization: The GCP security framework will handle authentication and access control, ensuring compliance and robust user data protection.
  7. Custom Frontend Interface: A custom-built frontend hosted on Firebase will serve as the primary user interface for apps, ensuring seamless integration with backend services.
  8. Standard Media APIs: Includes YouTube API and similar services to fetch data in compliance with platform guidelines.
  9. GitHub for Version Control: Used for managing the codebase, revisions, and deployment with GitHub Copilot as the main developer assistant.
  10. OpenAI: Used for metadata enrichment and internal AI-driven processing of media data.
  11. Centralized External API Calls:
  12. All calls to external APIs (e.g., YouTube API) are managed centrally by the admin or scheduled processes, avoiding unnecessary costs and complexity.
  13. External calls fetch as much data as possible (e.g., full descriptions, transcripts) to minimize redundant requests.

**3. Core Workflow**

* 1. Admin/Backend Workflow:
  2. API Query Management:
  3. Admin inputs filters for queries, including options like usePageToken to fetch results exceeding 50 documents.
  4. Queries fetch enriched data (e.g., keywords, descriptions, transcripts) from external APIs.
  5. AI-Driven Data Processing:
  6. Retrieved media metadata is analyzed and processed by AI to add value such as:
  7. Keyword Timestamps: Links to specific segments or chapters of the media content.
  8. Multi-Document Keys: Variations of documents saved with unique keys (e.g., videoID\_keyword\_keywordtimestamp).
  9. End-User Workflow:
  10. Custom Frontend Interface:
  11. Hosted on Firebase, this interface will display enriched metadata in a user-friendly feed.
  12. Users can fetch data in batches of 10 documents per scroll, ensuring a smooth and engaging experience.
  13. Scroll-Based Fetching:
  14. Users interact only with the local database, requesting metadata in small batches.
  15. Feed Display:
  16. Enriched metadata is displayed, with direct links to specific media segments.
  17. Defect Management:
  18. Open bugs and filter-related issues are tracked in a defect list for resolution.
  19. Example: Some filter options in the YouTube API are currently not working as intended.

**4. Architectural Highlights**

* 1. Flexibility and Reusability:
  2. The solution is a fully configurable template adaptable to any topic or niche by updating the keyword library.
  3. Designed to scale across multiple apps with no topic-specific hardcoding.
  4. AI Integration:
  5. OpenAI powers internal processing for metadata enrichment, creating added value in the database.
  6. Data Pipeline:
  7. External Fetching:
  8. Fetch metadata from YouTube API and similar sources based on admin-defined filters.
  9. Local Processing:
  10. AI processes fetched data to generate enriched metadata with additional layers of information.
  11. Staging and Storage:
  12. Store enriched metadata in Firebase Firestore for fast retrieval by end-user apps.
  13. Custom Frontend:
  14. The custom-built interface will:
  15. Showcase enriched data feeds.
  16. Offer seamless scrolling and fetching.
  17. Be fully hosted on Firebase for optimized performance.
  18. GCP Security and Authorization:
  19. Implement GCP’s security frameworks to:
  20. Authenticate users securely.
  21. Protect user data and comply with GDPR requirements.
  22. Leverage GCP’s role-based access control (RBAC) for sensitive admin tasks.

**5. Operational Policies**

* 1. Cost Management:
  2. APIs (e.g., OpenAI, YouTube) are configured to minimize unnecessary usage.
  3. APIs must never autonomously switch to higher-cost models without explicit approval from Mattias, including a clear justification.
  4. Security and Compliance:
  5. APIs interact only with approved services to ensure data privacy and integrity.
  6. No direct API requests from end-user apps to external services.

**6. Next Steps**

* 1. Setting Up API Roles:
  2. Define roles like "Markus" for architectural oversight and "Ulrich" for AI-driven metadata enrichment.
  3. Add a role for "Tomiko" to oversee frontend development.
  4. Custom Frontend Development:
  5. Design and build the user interface, focusing on usability and scalability.
  6. GCP Security Setup:
  7. Configure GCP authentication and authorization for secure access control.
  8. Defect Resolution:
  9. Address open bugs, especially related to YouTube API filters, before scaling.
  10. Testing and Iteration:
  11. Test the system end-to-end (E2E) with real user interactions, ensuring smooth data flow and seamless user experience.

This updated blueprint integrates the custom frontend and GCP security concept, ensuring the prototype aligns with long-term scalability and operational goals.

*From <*[*https://chatgpt.com/c/6741b697-f24c-800e-aa0b-95789c820db0*](https://chatgpt.com/c/6741b697-f24c-800e-aa0b-95789c820db0)*>*

Here’s a detailed technical breakdown of the youtubeApi.js file:

**Overview**

This file interacts with the YouTube Data API to fetch video metadata, details, and captions based on specified parameters. It provides three main functions:

* 1. fetchVideos: Fetches a list of videos based on channel ID, keyword, date, region, and other filters.
  2. fetchFullDescription: Retrieves detailed metadata for a specific video, including statistics and tags.
  3. fetchCaptions: Retrieves captions (transcriptions) for a specific video if available.

**Function Descriptions and Interlinking**

1. **fetchVideos**

* 1. Purpose: The main function used to fetch a list of videos from the YouTube API.
  2. Parameters:
  3. channelId: Restricts results to videos from a specific channel.
  4. keyword: Searches for videos containing the keyword in their title or description.
  5. publishedAfter: Filters videos published after a specific date.
  6. maxResults: Limits the number of videos retrieved (maximum 50 per page).
  7. regionCode: Restricts results to a specific geographic region.
  8. minViews: Filters videos with a minimum view count.
  9. usePagination: If true, fetches multiple pages of results using nextPageToken.
  10. Interlinking:
  11. Calls fetchFullDescription for each video to get detailed metadata.
  12. Utilizes fetchCaptions (indirectly, through video ID) if required by the downstream logic.
  13. Key Features:
  14. Uses pagination to retrieve results beyond the default page size (50).
  15. Filters videos in-memory to ensure the minViews requirement is respected.

2. **fetchFullDescription**

* 1. Purpose: Retrieves additional metadata for a specific video, including:
  2. Snippet details (e.g., title, description, tags).
  3. Content details (e.g., duration).
  4. Statistics (e.g., view count, like count, comment count).
  5. Parameters:
  6. videoId: The unique identifier for the video.
  7. Interlinking:
  8. Called by fetchVideos for every video returned by the initial search query.
  9. Key Features:
  10. Expands on the limited data provided by the initial video search.
  11. Ensures that enriched data (e.g., tags, view counts) is added to the final dataset.

3. **fetchCaptions**

* 1. Purpose: Fetches captions (transcriptions) for a specific video if available.
  2. Parameters:
  3. videoId: The unique identifier for the video.
  4. Interlinking:
  5. Not directly called in the current implementation of fetchVideos.
  6. Can be integrated into the batch job for enriching video metadata with captions.
  7. Key Features:
  8. Allows for the inclusion of subtitles or transcriptions in the final dataset.
  9. Useful for further processing, such as timestamping or keyword flagging.

**Interlinking Between Functions**

* 1. fetchVideos → fetchFullDescription:
  2. After retrieving the initial list of videos, fetchVideos uses fetchFullDescription to fetch detailed metadata for each video.
  3. fetchVideos → fetchCaptions (Potential):
  4. Although not currently implemented, captions could be fetched for each video as part of the enrichment process.
  5. Shared Parameters:
  6. All functions rely on videoId to identify specific videos for fetching additional details or captions.

**Key Features and Improvements**

**1. Batch Processing for Efficiency**

* 1. The fetchVideos function is designed to process multiple videos in batches. This improves API efficiency and ensures a consistent dataset for downstream applications.

**2. Pagination Handling**

* 1. The use of nextPageToken allows the function to retrieve large datasets that span multiple pages.
  2. The usePagination flag provides flexibility, enabling the admin to decide whether to fetch additional pages or stop after the first page.

**3. Metadata Enrichment**

* 1. By integrating fetchFullDescription and potentially fetchCaptions, the function ensures that the retrieved videos are enriched with:
  2. View count, like count, and comment count.
  3. Tags and category IDs.
  4. Duration and region-specific details.

**4. Custom Filters**

* 1. The inclusion of filters such as minViews and regionCode allows for highly targeted searches, ensuring that only relevant videos are retrieved.

**Current Issues and Suggestions**

* 1. Filter Functionality:
  2. Issue: Some filters (e.g., minViews) are implemented in-memory, which might lead to inefficiencies if a large number of videos are retrieved.
  3. Suggestion: Implement these filters at the query level if supported by the YouTube API.
  4. Caption Integration:
  5. Issue: Captions are not currently fetched as part of the main workflow.
  6. Suggestion: Integrate fetchCaptions into the enrichment pipeline to enhance metadata.
  7. Error Handling:
  8. Issue: Limited feedback for specific errors (e.g., invalid videoId or API quota exceeded).
  9. Suggestion: Improve error messages to make debugging easier.
  10. Scalability:
  11. Issue: The current setup might struggle with very large datasets or high traffic.
  12. Suggestion: Optimize API calls and consider caching mechanisms for frequently accessed data.

*From <*[*https://chatgpt.com/c/6741b697-f24c-800e-aa0b-95789c820db0*](https://chatgpt.com/c/6741b697-f24c-800e-aa0b-95789c820db0)*>*