
InMemoryDatabaseService

Brief Details

The `InMemoryDatabaseService` acts as a **mock database** for the Scheduler SDK, storing task definitions and webhook registrations in JavaScript `Map` objects. It's designed for **development and testing**, simulating persistence without a real database.

Role / Logic

This class implements the `IDatabaseService` interface. It provides basic CRUD (Create, Read, Update, Delete) operations for `tasks` and `webhookRegistrations` using in-memory `Map` objects. Methods like `init()`, `getTask()`, `saveTask()`, `updateTask()`, `deleteTask()`, `listAllTasks()`, `getWebhookRegistration()`, `saveWebhookRegistration()`, and `deleteWebhookRegistration()` manage these in-memory collections.

Future Scope

For **production environments**, `InMemoryDatabaseService` must be replaced with a real, **persistent database implementation**. A common and robust choice is **PostgreSQL**, often integrated using an ORM like **Prisma** (as indicated by your `schema.prisma` file).

Brief Steps to Implement PostgreSQL with Prisma:

1. **Install Prisma Client and CLI**:
2. Bash

```
npm install @prisma/client
npm install --save-dev prisma
```

3.
4.
5. **Configure `schema.prisma`**: Ensure your `schema.prisma` (which you already have) is correctly defined with `provider = "postgresql"` and your database connection string in `DATABASE_URL` environment variable.
6. **Generate Prisma Client**:
7. Bash

```
npx prisma generate
```

8.
9.
10. **Run Migrations**: Apply your schema to the database.

11. Bash

```
npx prisma migrate dev --name init
```

12.
13.
14. **Create a New Database Service Class**: Implement `IDatabaseService` in a new class (e.g., `PostgresDatabaseService`). This class would use the generated Prisma Client to interact with the PostgreSQL database.
15. TypeScript

```
// src/services/PostgresDatabaseService.ts (Conceptual)
import { PrismaClient } from '@prisma/client';
import { IDatabaseService, IngestionTaskDefinition, WebhookRegistryEntry } from
'../functions/ingestion/interfaces';
import { logger } from '@godspeedsystems/core';

export class PostgresDatabaseService implements IDatabaseService {
  private prisma: PrismaClient;

  constructor() {
    this.prisma = new PrismaClient();
  }

  async init(): Promise<void> {
    try {
      await this.prisma.$connect();
      logger.info("PostgresDatabaseService initialized and connected.");
    } catch (error) {
      logger.error("Failed to connect to PostgreSQL database.", error);
      throw error;
    }
  }

  // Implement all IDatabaseService methods using this.prisma
  async getTask(taskId: string): Promise<IngestionTaskDefinition | undefined> {
    const task = await this.prisma.ingestionTask.findUnique({ where: { id: taskId } });
    // You'll need to map Prisma's model to your IngestionTaskDefinition interface
    return task ? { ...task, definition: task.definition as any, lastRunStatus: task.lastRunStatus as
any } : undefined;
  }
  // ... implement other methods like saveTask, updateTask, etc.
  // Ensure proper error handling and data mapping.
}
```

16.
17.
18. **Inject the New Service**: In `GlobalIngestionLifecycleManager.getInstance()`, inject an instance of `PostgresDatabaseService` instead of `InMemoryDatabaseService` when running in a production-like environment.

Code Snippet of `InMemoryDatabaseService`

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleMana
ger.ts

// ... (imports)

// --- InMemoryDatabaseService (for local testing) ---
// This class implements IDatabaseService using in-memory Maps.
// In a production environment, you would replace this with a real database implementation.
class InMemoryDatabaseService implements IDatabaseService {
    private tasks: Map<string, IngestionTaskDefinition> = new Map();
    private webhookRegistrations: Map<string, WebhookRegistryEntry> = new Map();

    async init(): Promise<void> {
        logger.info("InMemoryDatabaseService initialized.");
        // No actual initialization needed for in-memory maps
    }

    async getTask(taskId: string): Promise<IngestionTaskDefinition | undefined> {
        return this.tasks.get(taskId);
    }

    async saveTask(task: IngestionTaskDefinition): Promise<void> {
        this.tasks.set(task.id, task);
        logger.debug(`InMemoryDatabaseService: Task '${task.id}' saved.`);
    }

    async updateTask(taskId: string, updates: Partial<IngestionTaskDefinition>): Promise<void> {
        const existingTask = this.tasks.get(taskId);
        if (existingTask) {
            Object.assign(existingTask, updates);
            logger.debug(`InMemoryDatabaseService: Task '${taskId}' updated.`);
        } else {
            logger.warn(`InMemoryDatabaseService: Attempted to update non-existent task '${taskId}'.`);
        }
```

```
    }

    async deleteTask(taskId: string): Promise<void> {
        this.tasks.delete(taskId);
        logger.debug(`InMemoryDatabaseService: Task '${taskId}' deleted.`);
    }

    async listAllTasks(): Promise<IngestionTaskDefinition[]> {
        return Array.from(this.tasks.values());
    }

    async getWebhookRegistration(sourceIdentifier: string): Promise<WebhookRegistryEntry |
undefined> {
        return this.webhookRegistrations.get(sourceIdentifier);
    }

    async saveWebhookRegistration(entry: WebhookRegistryEntry): Promise<void> {
        this.webhookRegistrations.set(entry.sourceIdentifier, entry);
        logger.debug(`InMemoryDatabaseService: Webhook registration for '${entry.sourceIdentifier}'
saved.`);
    }

    async updateWebhookRegistration(sourceIdentifier: string, updates:
Partial<WebhookRegistryEntry>): Promise<void> {
        const existingEntry = this.webhookRegistrations.get(sourceIdentifier);
        if (existingEntry) {
            Object.assign(existingEntry, updates);
            logger.debug(`InMemoryDatabaseService: Webhook registration for '${sourceIdentifier}'
updated.`);
        } else {
            logger.warn(`InMemoryDatabaseService: Attempted to update non-existent webhook
registration for '${sourceIdentifier}'.`);
        }
    }

    async deleteWebhookRegistration(sourceIdentifier: string): Promise<void> {
        this.webhookRegistrations.delete(sourceIdentifier);
        logger.debug(`InMemoryDatabaseService: Webhook registration for '${sourceIdentifier}'
deleted.`);
    }
}

// ... (rest of the GlobalIngestionLifecycleManager class)
```

GlobalIngestionLifecycleManager (Constructor and Static Instance)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (imports)

// --- GlobalIngestionLifecycleManager ---
interface DefaultCrawlerRegistryEntry {
    dataSource: new (...args: any[]) => GSDataSource;
    defaultTransformer: IngestionDataTransformer;
}

interface RegisteredPlugins {
    source: Map<string, { plugin: new (...args: any[]) => GSDataSource; transformer: IngestionDataTransformer }>;
    destination: Map<string, new (...args: any[]) => IDestinationPlugin>;
}

export class GlobalIngestionLifecycleManager extends EventEmitter implements IGlobalIngestionLifecycleManager {
    private registeredPlugins: RegisteredPlugins = { source: new Map(), destination: new Map() };
    private eventBus: EventEmitter = new EventEmitter();
    private dbService: IDatabaseService;

    private static instance: GlobalIngestionLifecycleManager;

    // FIX: Define the default crawler registry within the manager's scope
    private static readonly _defaultCrawlerRegistry: Record<string, DefaultCrawlerRegistryEntry> = {
        // These imports need to be at the top of the file
        'git-crawler': { dataSource: GitCrawlerDataSource, defaultTransformer: passthroughTransformer },
        'googledrive-crawler': { dataSource: GoogleDriveCrawlerDataSource, defaultTransformer: passthroughTransformer },
        'http-crawler': { dataSource: HttpCrawlerDataSource, defaultTransformer: htmlToPlaintextTransformer },
        // Add other default crawlers here as needed
    };

    private constructor(dbService?: IDatabaseService) {
        super();
```

```
        this.dbService = dbService || new InMemoryDatabaseService(); // Default to in-memory for local testing
        this.eventBus.on(IngestionEvents.TASK_COMPLETED, this.onTaskCompleted.bind(this));
        this.eventBus.on(IngestionEvents.TASK_FAILED, this.onTaskFailed.bind(this));
    }

    public static getInstance(dbService?: IDatabaseService): GlobalIngestionLifecycleManager {
        if (!GlobalIngestionLifecycleManager.instance) {
            GlobalIngestionLifecycleManager.instance = new GlobalIngestionLifecycleManager(dbService);
        } else if (dbService && GlobalIngestionLifecycleManager.instance.dbService instanceof InMemoryDatabaseService) {
            // Allow injecting a real DB service if currently using in-memory
            GlobalIngestionLifecycleManager.instance.setDatabaseService(dbService);
        }
        return GlobalIngestionLifecycleManager.instance;
    }

    // ... (rest of the GlobalIngestionLifecycleManager class)
}
```

Brief Details

This part of the `GlobalIngestionLifecycleManager` class handles its **initialization** and ensures it operates as a **singleton**. It also defines an **internal registry of default crawlers**.

Role / Logic

- **`private static readonly _defaultCrawlerRegistry`**: This is a `static readonly` property that holds a map of known crawler `pluginType` strings (like `'git-crawler'`) to their corresponding `DataSource` class constructors and default `IngestionDataTransformer` functions.
    - **Role**: It serves as the **centralized lookup table** for automatically registering default data sources when the Scheduler SDK starts or when the `sources()` method is called. Being `static` means it's shared across all instances (though there's only one due to singleton pattern), and `readonly` ensures it's initialized once and immutable.
- **`private constructor(dbService?: IDatabaseService)`**: This is the class constructor. It's `private` to enforce the singleton pattern.
    - **Input**: Optionally accepts an instance of `IDatabaseService`.
    - **Process**:
        1. Calls `super()` to initialize the `EventEmitter` base class.

2. Initializes `this.dbService`: If a `dbService` is provided, it uses that; otherwise, it defaults to `new InMemoryDatabaseService()`. This makes the Scheduler SDK usable out-of-the-box for development without a real database.
3. Registers internal event listeners: It listens for `TASK_COMPLETED` and `TASK_FAILED` events on its own `eventBus` to perform internal logging and state updates.

- **Role**: Sets up the fundamental state of the manager, including its database service and event handling.

- **`public static getInstance(dbService?: IDatabaseService): GlobalIngestionLifecycleManager`**: This is the **static factory method** to get the singleton instance of the `GlobalIngestionLifecycleManager`.
  - **Input**: Optionally accepts an instance of `IDatabaseService`.
  - **Process**:
    1. It checks if `GlobalIngestionLifecycleManager.instance` already exists.
    2. If not, it creates the single instance by calling `new GlobalIngestionLifecycleManager(dbService)`.
    3. If an instance already exists *and* a new `dbService` is provided (and the existing one is `InMemoryDatabaseService`), it allows injecting a real database service into the existing singleton. This is useful for testing or transitioning from an in-memory setup.
    4. Returns the single instance.
  - **Role**: Ensures that only one instance of the `GlobalIngestionLifecycleManager` exists throughout the application's lifetime, providing a central point of control.

Future Scope

- **Extensible `_defaultCrawlerRegistry`**: For very large or dynamic systems, the `_defaultCrawlerRegistry` could be loaded from an external configuration file or a plugin discovery mechanism, rather than being hardcoded in the class. This would allow new crawler types to be added without modifying the Scheduler SDK's code.
- **Robust Database Service Injection**: The `getInstance` method's logic for injecting a `dbService` could be made more explicit or managed by a dependency injection framework for complex production setups.
- **Type Safety for `_defaultCrawlerRegistry` Values**: While `DefaultCrawlerRegistryEntry` provides some type safety, ensuring that `dataSource` and `defaultTransformer` are always valid constructors/functions could be enhanced with more advanced TypeScript patterns or runtime checks if dynamic loading is introduced.

Let's continue analyzing the GlobalIngestionLifecycleManager.ts file. The next logical method in the sequence is setDatabaseService.

---

setDatabaseService (Public Method)

```
TypeScript

//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    public setDatabaseService(dbService: IDatabaseService): void {
        this.dbService = dbService;
        logger.info("GlobalIngestionLifecycleManager: Database service updated.");
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The setDatabaseService method allows for **dynamic injection or updating of the database service** used by the GlobalIngestionLifecycleManager.

Role / Logic

- **Input**: It takes one parameter: dbService, which must be an instance of IDatabaseService.
- **Process**: It simply assigns the provided dbService instance to the this.dbService private property of the manager. It also logs an informational message confirming the update.
- **Output**: This method does not return any value (void).
- **Significance**: This method is particularly useful in scenarios where the GlobalIngestionLifecycleManager might initially be instantiated with an InMemoryDatabaseService (for development/testing) but later needs to switch to a **real, persistent database service** (e.g., PostgresDatabaseService) during the application's lifecycle or for specific test setups. It's explicitly called within the getInstance static method to allow this "hot-swapping" from in-memory to a real database.

Future Scope

- **Validation**: Add validation to ensure the dbService provided is a valid, initialized instance, or to prevent changing the database service after the manager has already started processing tasks.

- **Lifecycle Management**: If the database service itself has a complex lifecycle (e.g., requiring explicit `disconnect()` calls), consider adding logic to manage the old `dbService` instance when a new one is set.
- **Dependency Injection Framework**: In larger applications, dependency injection frameworks (like InversifyJS, NestJS's DI) would typically handle this injection more robustly, abstracting away the direct `setDatabaseService` call.

`init` (Public Method)

```
TypeScript

//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    async init(): Promise<void> {
        logger.info("GlobalIngestionLifecycleManager initializing...");
        await this.dbService.init(); // Initialize the underlying database service
        logger.info("GlobalIngestionLifecycleManager initialized.");
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `init` method is responsible for the **initial setup and readiness** of the `GlobalIngestionLifecycleManager`. It's a crucial step that must be called before the manager can start processing tasks.

Role / Logic

- **Input**: This method takes no direct input parameters.
- **Process**:
    1. It logs an informational message indicating that the manager is starting its initialization process.
    2. It then calls `await this.dbService.init()`. This is a critical step because it delegates the initialization responsibility to the underlying database service (e.g., `InMemoryDatabaseService` or a future `PostgresDatabaseService`). For a real database, this would involve establishing a connection, verifying credentials, or setting up connection pools.
    3. Finally, it logs another informational message confirming that the manager has completed its initialization.

- **Output**: This method returns a `Promise<void>`, indicating that it completes asynchronously without returning a specific value.
- **Significance**: This method ensures that the manager's essential dependencies, particularly its database connection, are established and ready before any tasks are loaded or triggered. It separates the construction of the manager object from its operational readiness.

Future Scope

- **Configuration Validation**: Add more comprehensive validation of the overall manager configuration (e.g., ensuring all necessary plugins are registered) before proceeding with database initialization.
- **Dependency Health Checks**: Implement checks to ensure external dependencies beyond the database (e.g., message broker connections, external API client readiness) are also initialized and healthy.
- **Retry Mechanisms**: For real database connections, consider implementing retry logic with exponential backoff in `dbService.init()` to handle transient connection issues during startup.

`init` (Public Method)

TypeScript

```
// C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    async init(): Promise<void> {
        logger.info("GlobalIngestionLifecycleManager initializing...");
        await this.dbService.init(); // Initialize the underlying database service
        logger.info("GlobalIngestionLifecycleManager initialized.");
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `init` method is responsible for the **initial setup and readiness** of the `GlobalIngestionLifecycleManager`. It's a crucial step that must be called before the manager can start processing tasks.

Role / Logic

- **Input**: This method takes no direct input parameters.
- **Process**:
    1. It logs an informational message indicating that the manager is starting its initialization process.
    2. It then calls `await this.dbService.init()`. This is a critical step because it delegates the initialization responsibility to the underlying database service (e.g., `InMemoryDatabaseService` or a future `PostgresDatabaseService`). For a real database, this would involve establishing a connection, verifying credentials, or setting up connection pools.
    3. Finally, it logs another informational message confirming that the manager has completed its initialization.
- **Output**: This method returns a `Promise<void>`, indicating that it completes asynchronously without returning a specific value.
- **Significance**: This method ensures that the manager's essential dependencies, particularly its database connection, are established and ready before any tasks are loaded or triggered. It separates the construction of the manager object from its operational readiness.

Future Scope

- **Configuration Validation**: Add more comprehensive validation of the overall manager configuration (e.g., ensuring all necessary plugins are registered) before proceeding with database initialization.
- **Dependency Health Checks**: Implement checks to ensure external dependencies beyond the database (e.g., message broker connections, external API client readiness) are also initialized and healthy.
- **Retry Mechanisms**: For real database connections, consider implementing retry logic with exponential backoff in `dbService.init()` to handle transient connection issues during startup.

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

  async stop(): Promise<void> {
    logger.info("GlobalIngestionLifecycleManager stopping. Clearing internal states.");
    // No cron jobs to destroy here as they are managed externally by Godspeed.
```

```
        // Webhooks are assumed to persist externally, so no mass deregistration here.
        // Individual deregistration happens on task deletion/disabling.
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

`stop` (Public Method)

Brief Details

The `stop` method is called to **gracefully shut down** the `GlobalIngestionLifecycleManager`. Its primary role is to log the cessation of operations and acknowledge the state of managed resources.

Role / Logic

- **Input**: None.
- **Process**:
    1. It logs an informational message indicating that the manager is stopping and clearing internal states.
    2. It explicitly notes that no cron jobs are destroyed here because they are managed externally by the Godspeed framework.
    3. It also clarifies that no mass deregistration of webhooks occurs, as webhooks are assumed to persist externally and individual deregistration is handled when tasks are deleted or disabled.
- **Output**: Returns a `Promise<void>`.
- **Significance**: This method serves as a **clean shutdown point**. In this simulated environment, it primarily provides logging. In a real-world scenario, it would be crucial for releasing resources (e.g., closing database connections, unsubscribing from message queues) that were opened during `init` or `start`.

Future Scope

- **Resource Release**: Implement logic to explicitly close database connections (`this.dbService.$disconnect()` if using Prisma), shut down any internal timers, or release other resources acquired by the manager.
- **Graceful Shutdown of Active Tasks**: If tasks could be long-running, consider adding logic to gracefully stop or signal active tasks to complete before the manager fully shuts down.
- **State Saving**: For non-persistent database services (like the in-memory one), consider adding a mechanism to serialize and save the current state before stopping, allowing for quick restoration on restart (though this might contradict the "simulated" nature).

```
TypeScript

//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    public getEventBus(): EventEmitter {
        return this.eventBus;
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

`getEventBus` (Public Method)

Brief Details

The `getEventBus` method provides **access to the internal event emitter** of the `GlobalIngestionLifecycleManager`.

Role / Logic

- **Input**: None.
- **Process**: It simply returns the `this.eventBus` (an `EventEmitter` instance) that was initialized in the constructor.
- **Output**: Returns an `EventEmitter` instance.
- **Significance**: This method allows other parts of the application (e.g., `final-test.ts` for debugging, or other modules that need to react to task lifecycle events) to **subscribe to events** emitted by the `GlobalIngestionLifecycleManager` (like `TASK_COMPLETED`, `TASK_FAILED`, `DATA_TRANSFORMED`). This is crucial for the system's observability and for building reactive workflows.

Future Scope

- **Typed Events**: For enhanced type safety, consider using a more strongly typed event emitter library or pattern (e.g., `TypedEventEmitter`) to define specific event names and their payload types.
- **Event Filtering/Routing**: For very complex systems, a more advanced event bus might offer features like event filtering, routing, or persistence, but for this SDK's current scope, a direct `EventEmitter` is sufficient.

Here's the combined explanation for `registerSource`, `registerDestination`, and `sources` methods.

`registerSource`, `registerDestination`, and `sources` (Public Methods)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleMana
ger.ts

// ... (previous code)

    public registerSource(pluginType: string, sourcePlugin: new (...args: any[]) => GSDataSource,
transformer: IngestionDataTransformer): void {
        this.registeredPlugins.source.set(pluginType, { plugin: sourcePlugin, transformer });
        logger.info(`Source plugin '${pluginType}' registered.`);
    }

    public registerDestination(pluginType: string, destinationPlugin: new (...args: any[]) =>
IDestinationPlugin): void {
        this.registeredPlugins.destination.set(pluginType, destinationPlugin);
        logger.info(`Destination plugin '${pluginType}' registered.`);
    }

    /**
     * Allows registering multiple default data sources by providing an array of their names.
     * This method looks up the DataSource class and default transformer from an internal registry.
     * @param crawlerTypes An array of string names for the crawler types to register (e.g.,
['git-crawler', 'googledrive-crawler']).
     */
    public async sources(crawlerTypes: string[]): Promise<void> {
        logger.info("Registering default data sources from array...");
        for (const type of crawlerTypes) {
            const entry = GlobalIngestionLifecycleManager._defaultCrawlerRegistry[type];
            if (entry) {
                this.registerSource(type, entry.dataSource, entry.defaultTransformer);
                logger.info(`Registered '${type}' source with its default transformer.`);
            } else {
                logger.warn(`Attempted to register unknown crawler type: '${type}'. Skipping.`);
            }
        }
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

These methods are responsible for **registering available data source (crawler) and data destination plugins** with the `GlobalIngestionLifecycleManager`. They make the plugins known to the system so they can be dynamically instantiated and used when a task is executed. The `sources` method provides a streamlined way to register multiple default crawlers.

Role / Logic

- **`registerSource(pluginType, sourcePlugin, transformer)`**:
    - **Role**: Manually registers a single data source plugin.
    - **Logic**: Takes a `pluginType` string (e.g., `'git-crawler'`), the `DataSource` class constructor, and a default `IngestionDataTransformer` function. It stores this information in the `this.registeredPlugins.source` Map.
- **`registerDestination(pluginType, destinationPlugin)`**:
    - **Role**: Manually registers a single data destination plugin.
    - **Logic**: Takes a `pluginType` string (e.g., `'file-system-destination'`) and the `IDestinationPlugin` class constructor. It stores this in the `this.registeredPlugins.destination` Map.
- **`sources(crawlerTypes)`**:
    - **Role**: Provides a convenient way to automatically register multiple default data sources.
    - **Logic**: Accepts an array of `pluginType` strings (e.g., `['git-crawler', 'http-crawler']`). It iterates through this array, looks up each type in the internal `_defaultCrawlerRegistry` (a static map defined in the class), and if found, calls `this.registerSource()` internally with the corresponding `DataSource` class and its default `transformer`. If a type is not found in the registry, it logs a warning.
- **Overall Role**: These methods populate the internal registries that `runOrchestrator` uses to dynamically create instances of crawlers and destinations based on a task's configuration.

Future Scope

- **Plugin Discovery**: For larger systems, these methods could be part of an automated plugin discovery mechanism (e.g., scanning a directory for plugins) rather than requiring explicit calls or a hardcoded registry.
- **Version Management**: Implement a way to register multiple versions of the same plugin type.
- **Dependency Injection**: Integrate with a more formal dependency injection container to manage plugin instances and their dependencies.
- **Health Checks**: Add a mechanism to perform basic health checks or validation of the registered plugins at registration time.

---

scheduleTask (Public Method)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleMana
ger.ts

// ... (previous code)

    async scheduleTask(taskDefinition: IngestionTaskDefinition): Promise<GSStatus> {
        const taskId: string = taskDefinition.id || uuidv4();
        if (await this.dbService.getTask(taskId)) {
            logger.warn(`Task '${taskId}' already exists. Use updateTask to modify.`);
            return new GSStatus(false, 409, `Task '${taskId}' already exists.`);
        }

        const taskToSave: IngestionTaskDefinition = {
            ...taskDefinition, // Create a new object to avoid direct mutation of the input
            id: taskId,
            currentStatus: IngestionTaskStatus.SCHEDULED,
            lastRun: undefined,
            lastRunStatus: undefined,
        };

        await this.dbService.saveTask(taskToSave);
        this.eventBus.emit(IngestionEvents.TASK_SCHEDULED, taskToSave);
        logger.info(`Task '${taskToSave.name}' (${taskToSave.id}) scheduled.`);

        if (taskToSave.enabled) {
            const trigger = taskToSave.trigger as IngestionTrigger;
            if (trigger.type === 'cron') {
                logger.info(`Task '${taskToSave.id}' is configured for Godspeed Cron trigger "${(trigger as
CronTrigger).expression}".`);
            } else if (trigger.type === 'webhook') {
                const registrationStatus = await this.registerWebhook(taskToSave);
                if (!registrationStatus.success) {
                    logger.error(`Failed to register webhook for task ${taskToSave.id}. Task might not be
active.`);
                    await this.dbService.updateTask(taskToSave.id, { currentStatus:
IngestionTaskStatus.FAILED, lastRunStatus: registrationStatus });
                    return registrationStatus;
                }
            }
```

```
    }
    return new GSStatus(true, 200, "Task scheduled successfully.");
  }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `scheduleTask` method is responsible for **registering a new ingestion task** with the Scheduler SDK. It handles task ID generation, checks for task uniqueness, persists the task definition to the database, and initiates webhook registration if applicable.

Role / Logic

- **Input**: `taskDefinition` (an `IngestionTaskDefinition` object) which contains all details of the task.
- **Process**:
    1. **Generate Task ID**: If `taskDefinition.id` is not provided, a unique `taskId` is generated using `uuidv4()`.
    2. **Check Uniqueness**: It calls `this.dbService.getTask(taskId)` to **check if a task with the same ID already exists**. If it does, it logs a warning and returns a `GSStatus` with a `409 Conflict` code, preventing duplicate task creation.
    3. **Prepare Task for Saving**: A new `taskToSave` object is created as a **mutable copy** of the input `taskDefinition`. Its `id` is set, and `currentStatus` is initialized to `SCHEDULED`. `lastRun` and `lastRunStatus` are cleared.
    4. **Persist Task**: `this.dbService.saveTask(taskToSave)` is called to save the task definition to the database.
    5. **Emit Event**: A `TASK_SCHEDULED` event is emitted on the `eventBus`.
    6. **Handle Enabled Tasks**: If `taskToSave.enabled` is `true`:
        - For **cron tasks**, it logs a message about the cron trigger being configured.
        - For **webhook tasks**, it calls `this.registerWebhook(taskToSave)`. If this registration fails, it logs an error, updates the task's status to `FAILED` in the database, and returns the failure status.
    7. **Return Status**: Finally, it returns a `GSStatus(true, 200)` indicating successful scheduling.
- **Output**: Returns a `Promise<GSStatus>`.

Future Scope

- **Input Validation**: Add more robust validation for the `taskDefinition` schema (e.g., using a JSON Schema validator) to ensure all required fields are present and correctly formatted before saving.

- **Transactionality**: If using a real database, ensure `saveTask` and `registerWebhook` operations are part of a single transaction to maintain data consistency.
- **Asynchronous Webhook Registration**: For high-volume task scheduling, webhook registration could be made asynchronous (e.g., by placing a message on a queue) to prevent blocking the `scheduleTask` method.
- **Detailed Error Messages**: Improve the error messages returned for specific validation failures.

`updateTask` (Public Method)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    async updateTask(taskId: string, updates: Partial<IngestionTaskDefinition>): Promise<GSStatus>
{
        const existingTask = await this.dbService.getTask(taskId);
        if (!existingTask) {
            return new GSStatus(false, 404, `Task with ID ${taskId} not found.`);
        }

        const oldTask = JSON.parse(JSON.stringify(existingTask)) as IngestionTaskDefinition;
        const updatedTask = { ...existingTask, ...updates };

        await this.dbService.updateTask(taskId, updatedTask);
        this.eventBus.emit(IngestionEvents.TASK_UPDATED, updatedTask);
        logger.info(`Task '${taskId}' updated.`);

        // Handle webhook changes
        if (oldTask.trigger.type === 'webhook' || updatedTask.trigger.type === 'webhook') {
            if (oldTask.trigger.type === 'webhook' && updatedTask.trigger.type !== 'webhook') {
                // Scenario 1: Task changed from webhook to non-webhook
                await this.deregisterWebhook(oldTask.id);
            } else if (updatedTask.trigger.type === 'webhook') {
                // Scenario 2: Task is still webhook or changed to webhook
                const oldWebhookTrigger = oldTask.trigger as WebhookTrigger;
                const newWebhookTrigger = updatedTask.trigger as WebhookTrigger;

                const oldSourceIdentifier = this.getSourceIdentifier(oldTask.source.pluginType,
oldTask.source.config);
                const newSourceIdentifier = this.getSourceIdentifier(updatedTask.source.pluginType,
updatedTask.source.config);
```

```
            // If source identifier changed or externalWebhookId is missing (implies re-registration
needed)
            if (oldSourceIdentifier !== newSourceIdentifier || !newWebhookTrigger.externalWebhookId)
{
                if (oldSourceIdentifier) {
                    await this.deregisterWebhook(oldTask.id); // Deregister old webhook if source
changed
                }
                if (updatedTask.enabled) {
                    await this.registerWebhook(updatedTask); // Register new webhook if enabled
                }
            } else if (updates.enabled === false) {
                // Scenario 3: Webhook task disabled
                await this.deregisterWebhook(taskId);
            } else if (updates.enabled === true && !newWebhookTrigger.externalWebhookId) {
                // Scenario 4: Webhook task enabled without external ID (should trigger registration)
                await this.registerWebhook(updatedTask);
            }
        }
    }

    return new GSStatus(true, 200, "Task updated successfully.");
  }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `updateTask` method modifies an existing ingestion task's definition and status in the Scheduler SDK's database. It also intelligently handles changes related to webhook triggers, ensuring external webhook registrations are correctly managed (deregistered or re-registered) based on the update.

Role / Logic

- **Input**: `taskId` (string, the ID of the task to update) and `updates` (`Partial<IngestionTaskDefinition>`, an object containing the fields to change).
- **Process Flow**:
    1. **Retrieve Existing Task**: It first fetches the `existingTask` from `dbService` using `taskId`. If not found, it returns a `404 Not Found` status.

2. **Create Copies**: It creates a deep copy of the existingTask (as oldTask) and a merged updatedTask object. This is crucial as configuration objects from node-config are immutable.
3. **Persist Update**: The updatedTask is saved to the database via dbService.updateTask().
4. **Emit Event**: A TASK_UPDATED event is emitted on the eventBus.
5. **Handle Webhook Changes**: This is the most complex part, ensuring consistency with external webhook services:
    - **Type Change (Webhook to Non-Webhook)**: If the oldTask was a webhook but updatedTask is not, it calls deregisterWebhook(oldTask.id).
    - **Source Identifier Change / Missing External ID**: If the sourceIdentifier (e.g., repo URL, folder ID) of the task changes, or if externalWebhookId is missing from the newWebhookTrigger (implying the external webhook needs to be re-registered), it first deregisterWebhooks the old one (if oldSourceIdentifier exists) and then registerWebhooks the updatedTask (if it's enabled).
    - **Disable Webhook Task**: If the task is explicitly disabled (updates.enabled === false), it calls deregisterWebhook(taskId).
    - **Enable Webhook Task (without external ID)**: If the task is enabled and externalWebhookId is missing (e.g., re-enabling a task whose external webhook was lost), it calls registerWebhook(updatedTask).
6. **Return Status**: Returns a GSStatus(true, 200) on successful update.

- **Output**: Returns a Promise<GSStatus>.

Future Scope

- **Optimized Webhook Re-registration**: The current logic might re-register webhooks more often than strictly necessary (e.g., if only non-webhook-related task properties change). Future improvements could involve more granular checks to only re-register if callbackurl, credentials, or endpointId of the webhook trigger actually change.
- **Transactionality**: For a real database, ensure the database updates (dbService.updateTask) and external API calls (registerWebhook/deregisterWebhook) are managed within a transaction to maintain atomicity and consistency.
- **Partial Updates for Webhook Registration**: When registeredTasks is updated, dbService.updateWebhookRegistration is called. Ensure this method handles partial updates to the array efficiently in a real database (e.g., using atomic array operations if supported, or careful read-modify-write).
- **Asynchronous Webhook Operations**: For performance, especially with many updates, external registerWebhook/deregisterWebhook calls could be offloaded to an asynchronous queue.

`enableTask` (Public Method)

```
TypeScript

//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    async enableTask(taskId: string): Promise<GSStatus> {
        const task = await this.dbService.getTask(taskId);
        if (!task) return new GSStatus(false, 404, `Task with ID ${taskId} not found.`);
        if (task.enabled) return new GSStatus(true, 200, "Task is already enabled.");

        const updates = { enabled: true };
        return this.updateTask(taskId, updates);
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `enableTask` method **activates an existing ingestion task**, allowing it to be triggered by its configured schedule or webhook.

Role / Logic

- **Input**: `taskId` (string, the ID of the task to enable).
- **Process**:
    1. **Retrieve Task**: It fetches the task from `this.dbService.getTask(taskId)`. If the task is not found, it returns a `404 Not Found` status.
    2. **Check Current Status**: If the task is already `enabled`, it returns a `200 OK` status, indicating no action is needed.
    3. **Update Task**: It then calls `this.updateTask(taskId, { enabled: true })` to change the task's status in the database. This leverages the `updateTask` method's existing logic, which also handles any necessary webhook re-registrations if the task is webhook-triggered.
- **Output**: Returns a `Promise<GSStatus>`.

Future Scope

- **Detailed Status**: Provide more specific messages if the task cannot be enabled due to underlying issues (e.g., invalid configuration).

- **Event Emission**: Emit a `TASK_ENABLED` event after successful enablement, which could be useful for monitoring or other reactive processes.

`disableTask` (Public Method)

```
TypeScript

//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    async disableTask(taskId: string): Promise<GSStatus> {
        const task = await this.dbService.getTask(taskId);
        if (!task) return new GSStatus(false, 404, `Task with ID ${taskId} not found.`);
        if (!task.enabled) return new GSStatus(true, 200, "Task is already disabled.");

        const updates = { enabled: false };
        return this.updateTask(taskId, updates);
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `disableTask` method **deactivates an existing ingestion task**, preventing it from being triggered by its configured schedule or webhook.

Role / Logic

- **Input**: `taskId` (string, the ID of the task to disable).
- **Process**:
    1. It retrieves the task from `this.dbService.getTask(taskId)`. If the task is not found, it returns a `404 Not Found` status.
    2. If the task is already `disabled`, it returns a `200 OK` status, indicating no action is needed.
    3. It then calls `this.updateTask(taskId, { enabled: false })` to change the task's status in the database. This leverages the `updateTask` method's existing logic, which handles any necessary webhook deregistration if the task is webhook-triggered and this is the last active task associated with that webhook.
- **Output**: Returns a `Promise<GSStatus>`.

Future Scope

- **Event Emission**: Emit a `TASK_DISABLED` event after successful disablement, which could be useful for monitoring or other reactive processes.
- **Forced Termination**: If a task is currently `RUNNING`, consider adding logic to attempt to gracefully terminate its execution before marking it as disabled.

TypeScript

```
// C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    async deleteTask(taskId: string): Promise<GSStatus> {
        const task = await this.dbService.getTask(taskId);
        if (!task) return new GSStatus(false, 404, `Task with ID ${taskId} not found.`);

        // Deregister webhook if this was the last task using it.
        const trigger = task.trigger as IngestionTrigger;
        let res
        if (trigger.type === 'webhook') {
            res = await this.deregisterWebhook(taskId);
        }
        if(!res?.success){
            return new GSStatus(false, 403, "error in deleting task");
        }
        await this.dbService.deleteTask(taskId);
        this.eventBus.emit(IngestionEvents.TASK_DELETED, taskId);
        logger.info(`Task '${taskId}' deleted.`);

        return new GSStatus(true, 200, "Task deleted successfully.");
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

`deleteTask` (Public Method)

Brief Details

The deleteTask method is responsible for **permanently removing an ingestion task** from the Scheduler SDK's management. It ensures that associated external webhooks are also deregistered if the deleted task was the last one using that webhook.

Role / Logic

- **Input**: taskId (string, the ID of the task to delete).
- **Process**:
    1. **Retrieve Task**: It first fetches the task from this.dbService.getTask(taskId). If the task is not found, it returns a 404 Not Found status.
    2. **Deregister Webhook (if applicable)**:
        - It checks if the task's trigger.type is 'webhook'.
        - If so, it calls this.deregisterWebhook(taskId). This method handles the logic of removing the taskId from the registeredTasks array of the associated WebhookRegistration record. **Crucially, it will only perform the external webhook deregistration (with GitHub/Google Drive) if the registeredTasks array becomes empty after this task's ID is removed.**
        - If deregisterWebhook fails (e.g., due to external API issues), it returns a 403 Forbidden status, preventing the task from being deleted internally until the webhook issue is resolved.
    3. **Delete Task from DB**: If webhook deregistration (or its check) is successful, it proceeds to delete the task definition from the database via this.dbService.deleteTask(taskId).
    4. **Emit Event**: A TASK_DELETED event is emitted on the eventBus.
    5. **Log & Return Status**: It logs an informational message and returns a GSStatus(true, 200) indicating successful deletion.
- **Output**: Returns a Promise<GSStatus>.

Future Scope

- **Confirmation Prompt**: In a real application, a user interface might require a confirmation step before permanent deletion.
- **Archiving**: Instead of outright deletion, consider an archiving mechanism for tasks to retain historical data.
- **Cascade Deletion**: If any other entities are directly dependent on a task (e.g., associated logs, metrics), ensure they are also cleaned up or archived.
- **Error Handling Granularity**: Provide more specific error messages if the deregisterWebhook call fails, detailing the reason for the 403 status.

`getTask` (Public Method)

```TypeScript
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

  async getTask(taskId: string): Promise<IngestionTaskDefinition | undefined> {
    return this.dbService.getTask(taskId);
  }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `getTask` method retrieves a **single, specific ingestion task definition** from the Scheduler SDK's database.

Role / Logic

- **Input**: `taskId` (string, the unique identifier of the task to retrieve).
- **Process**: It directly calls `this.dbService.getTask(taskId)`, delegating the actual database lookup to the configured database service.
- **Output**: Returns a `Promise` that resolves to an `IngestionTaskDefinition` object if the task is found, or `undefined` if it does not exist.
- **Significance**: This method is a fundamental **read operation** for the Scheduler. It's used internally by other manager methods (like `updateTask`, `deleteTask`, `triggerManualTask`) to access task details.

Future Scope

- **Caching**: For frequently accessed tasks, consider implementing a caching layer (e.g., Redis) to reduce database load and improve retrieval performance.
- **Error Handling**: While `dbService.getTask` might handle its own errors, `getTask` could add specific logging or re-throw custom errors for better debugging.

`listTasks` (Public Method)

```TypeScript
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

  async listTasks(): Promise<IngestionTaskDefinition[]> {
    return this.dbService.listAllTasks();
  }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `listTasks` method retrieves **all currently defined ingestion tasks** from the Scheduler SDK's database.

Role / Logic

- **Input**: None.
- **Process**: It directly calls `this.dbService.listAllTasks()`, delegating the retrieval of all task definitions to the underlying database service.
- **Output**: Returns a `Promise` that resolves to an array of `IngestionTaskDefinition` objects.
- **Significance**: This method is a fundamental **read operation** used by various parts of the Scheduler, such as the `start()` method (to re-schedule tasks on startup) and `triggerAllEnabledCronTasks()` (to find tasks that are due). It also provides an interface for external monitoring or management tools to list all configured tasks.

Future Scope

- **Pagination**: For systems with a large number of tasks, implement pagination (e.g., `limit`, `offset`) to avoid loading all tasks into memory at once.
- **Filtering/Sorting**: Add options to filter tasks by status, type, or other criteria, and to sort the results.
- **Caching**: For frequently accessed lists, consider a caching layer to improve performance.

`triggerManualTask` (Public Method)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleMana
ger.ts

// ... (previous code)

   async triggerManualTask(ctx: GSContext, taskId: string, initialPayload?: any):
Promise<GSStatus> {
       const task = await this.dbService.getTask(taskId);
       if (!task) {
           const msg = `Task with ID ${taskId} not found.`;
           logger.error(msg);
           return new GSStatus(false, 404, msg);
       }
       if (!task.enabled) {
           const msg = `Task with ID ${taskId} is disabled and cannot be triggered manually.`;
           logger.warn(msg);
           return new GSStatus(false, 403, msg);
       }

       const sourceIdentifier = this.getSourceIdentifier(task.source.pluginType, task.source.config);
       if (sourceIdentifier) {
           const webhookEntry = await this.dbService.getWebhookRegistration(sourceIdentifier);
           if (webhookEntry) {
               initialPayload = {
                   ...initialPayload,
                   startPageToken: webhookEntry.startPageToken,
                   nextPageToken: webhookEntry.nextPageToken,
                   otherCrawlerSpecificTokens: webhookEntry.otherCrawlerSpecificTokens
               };
           }
       }

       return this.runOrchestrator(ctx, task, initialPayload);
   }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The triggerManualTask method allows an **administrator or user to initiate an ingestion task on demand**. It acts as an immediate, one-off trigger for a specific task, bypassing cron schedules or webhook events.

Role / Logic

- **Input**: ctx (Godspeed context), taskId (string, the ID of the task to trigger), and an optional initialPayload (any, additional data to pass to the crawler).
- **Process**:
    1. **Task Retrieval**: It first attempts to retrieve the task definition from this.dbService.getTask(taskId). If the task is not found, it logs an error and returns a 404 Not Found status.
    2. **Enablement Check**: It verifies if the retrieved task is enabled. If the task is disabled, it logs a warning and returns a 403 Forbidden status, preventing execution.
    3. **Payload Enrichment**: If the task's source has a sourceIdentifier (meaning it's a type that might have associated webhook registrations, like Git or Google Drive), it attempts to retrieve the corresponding webhookEntry from this.dbService.getWebhookRegistration(). If found, it **enriches the initialPayload** with any stored continuation tokens (like startPageToken, nextPageToken, otherCrawlerSpecificTokens) from that webhookEntry. This ensures manual triggers can also benefit from incremental processing.
    4. **Orchestrator Invocation**: Finally, it delegates the actual task execution to this.runOrchestrator(ctx, task, initialPayload), which will instantiate and run the appropriate crawler.
- **Output**: Returns a Promise<GSStatus> reflecting the outcome of the task execution.

Future Scope

- **Authentication/Authorization**: For a production system, this method would typically require **authentication and authorization checks** to ensure only authorized users can trigger tasks.
- **Asynchronous Triggering**: For long-running tasks, consider making the manual trigger asynchronous (e.g., by placing a message on a queue) to prevent the API request from timing out.
- **Detailed initialPayload Validation**: Add validation for the structure and content of the initialPayload if it's expected to contain specific data for certain tasks.
- **Concurrency Control**: Implement mechanisms to prevent a manual trigger from running concurrently with an already active cron or webhook trigger for the same task, if that's an undesired behavior.

`triggerWebhookTask` (Public Method)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleMana
ger.ts

// ... (previous code)

   async triggerWebhookTask(ctx: GSContext, endpointId: string, rawRequest: any, requestHeaders:
any): Promise<GSStatus> {
       logger.info(`Webhook received for endpoint ID: ${endpointId}.`);

       // 1. Get all enabled webhook tasks for this endpoint (initial broad filter)
       const allWebhookTasksForEndpoint = await this.dbService.listAllTasks();
       const tasksMatchingEndpoint = allWebhookTasksForEndpoint.filter(
           t => t.enabled && t.trigger.type === 'webhook' && (t.trigger as WebhookTrigger).endpointId
=== endpointId
       );
       logger.info(`tasksMatchingEndpoint.length:${tasksMatchingEndpoint.length}`)
       if (tasksMatchingEndpoint.length === 0) {
           const msg = `(from trg webhook)No enabled tasks found for webhook endpoint ID:
${endpointId}.`;
           logger.warn(msg);
           return new GSStatus(false, 404, msg);
       }

       let firstStatus: GSStatus | undefined;
       let initialProcessedResult: ProcessedWebhookResult;
       let finalProcessedResult: ProcessedWebhookResult;

       try {
           // Determine the webhook service type from the first matching task
           const webhookService = tasksMatchingEndpoint[0].source.pluginType;

           // 2. Preliminary processing to extract externalResourceId and payload (without secret
validation yet)
           initialProcessedResult = processWebhookRequest(webhookService, requestHeaders,
undefined, rawRequest);

           if (!initialProcessedResult.isValid) {
               const msg = initialProcessedResult.error || `Preliminary webhook processing failed for
endpoint ${endpointId}.`;
               logger.error(msg, { payload: initialProcessedResult.payload });
               return new GSStatus(false, 400, msg);
```

```
        }

        if (!initialProcessedResult.externalResourceId) {
            const msg = `Webhook processed successfully but could not extract externalResourceId
for endpoint ${endpointId}.`;
            logger.error(msg, { payload: initialProcessedResult.payload });
            return new GSStatus(false, 400, msg);
        }

        // 3. Find the specific WebhookRegistryEntry using the extracted externalResourceId
        const webhookEntry = await
this.dbService.getWebhookRegistration(initialProcessedResult.externalResourceId);

        if (!webhookEntry) {
            const msg = `No webhook registration found for resource
'${initialProcessedResult.externalResourceId}' linked to endpoint '${endpointId}'.`;
            logger.warn(msg);
            // It's a valid webhook, just no task configured for this specific resource.
            return new GSStatus(true, 200, msg);
        }

        // 4. Perform signature validation using the correct secret from the registry entry
        let tokenForValidation: string | undefined;
        if (webhookService === 'gdrive-crawler') {
            tokenForValidation = webhookEntry.externalWebhookId; // For GDrive, x-goog-channel-id
is externalWebhookId
        } else {
            tokenForValidation = webhookEntry.secret; // For Git, x-hub-signature uses the secret
        }

        finalProcessedResult = processWebhookRequest(webhookService, requestHeaders,
tokenForValidation, rawRequest);

        if (!finalProcessedResult.isValid) {
            const msg = finalProcessedResult.error || "Webhook request signature validation failed.";
            logger.error(msg);
            return new GSStatus(false, 401, msg);
        }

        // Ensure the resource ID is still consistent after full validation (should be)
        if (finalProcessedResult.externalResourceId !== initialProcessedResult.externalResourceId) {
            const msg = `Resource ID mismatch after full webhook validation. Initial:
${initialProcessedResult.externalResourceId}, Final: ${finalProcessedResult.externalResourceId}`;
            logger.error(msg);
            return new GSStatus(false, 500, msg);
        }
```

```
            // 5. Filter tasks based on the task IDs registered with this specific webhookEntry
            const tasksToTriggerForResource = tasksMatchingEndpoint.filter(task =>
                webhookEntry.registeredTasks.includes(task.id)
            );

            if (tasksToTriggerForResource.length === 0) {
                const msg = `Webhook registration for resource
'${initialProcessedResult.externalResourceId}' exists, but no enabled tasks are currently linked to it.`;
                logger.warn(msg);
                return new GSStatus(true, 200, msg);
            }

            // 6. Iterate and trigger only the relevant tasks
            for (const task of tasksToTriggerForResource) {
                // Re-fetch task to ensure latest state, though webhookEntry.registeredTasks should be
authoritative
                const taskToExecute = await this.dbService.getTask(task.id);
                if (!taskToExecute) {
                    logger.warn(`Task '${task.id}' found in webhook registry but not in DB. Skipping.`);
                    continue;
                }

                const sourceIdentifier = this.getSourceIdentifier(taskToExecute.source.pluginType,
taskToExecute.source.config);
                let currentWebhookState: WebhookRegistryEntry | undefined;
                if (sourceIdentifier) {
                    currentWebhookState = await this.dbService.getWebhookRegistration(sourceIdentifier);
                }

                const initialPayload = {
                    taskDefinition: taskToExecute, // Use the re-fetched task
                    webhookPayload: finalProcessedResult.payload, // Use the fully processed payload
                    externalResourceId: finalProcessedResult.externalResourceId, // Pass the extracted
resource ID
                    changeType: finalProcessedResult.changeType, // Pass the extracted change type
                    startPageToken: currentWebhookState?.startPageToken,
                    nextPageToken: currentWebhookState?.nextPageToken,
                    otherCrawlerSpecificTokens: currentWebhookState?.otherCrawlerSpecificTokens
                };
                const status = await this.runOrchestrator(ctx, taskToExecute, initialPayload);
                if (!firstStatus) {
                    firstStatus = status; // Capture the status of the first task
                }
            }
```

```
            return firstStatus || new GSStatus(false, 500, "Webhook could not be triggered due to an unknown error.");

        } catch (err: any) {
            const msg = `Error processing webhook payload: ${err.message}`;
            logger.error(msg, { error: err });
            return new GSStatus(false, 500, msg);
        }
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

This method is the **entry point for all incoming webhook notifications** to the Scheduler SDK. Its role is to robustly validate the webhook, identify which ingestion tasks are associated with it, and then trigger their execution.

Role / Logic

- **Input**:
    1. `ctx`: The Godspeed context, containing the raw HTTP `requestHeaders` and `rawRequest` (body) of the incoming webhook.
    2. `endpointId`: A string representing the local API endpoint that received the webhook (e.g., `/webhook/github/`).
    3. `rawRequest`: The raw HTTP body of the webhook.
    4. `requestHeaders`: The raw HTTP headers of the webhook.
- **Process Flow**:
    1. **Initial Task Filtering**: It first retrieves all `enabled` webhook tasks from the database that are configured to listen on the given `endpointId`. If no such tasks are found, it returns a `404 Not Found` status.
    2. **Preliminary Webhook Processing**: It calls **`processWebhookRequest(webhookService, requestHeaders, undefined, rawRequest)`** (from the Crawler SDK utilities). This first pass extracts the `externalResourceId` (e.g., GitHub repo URL, Google Drive folder ID) from the webhook payload without performing secret validation yet. This is crucial because the `externalResourceId` is needed to look up the correct secret in the database. If this preliminary step fails or `externalResourceId` cannot be extracted, it returns a `400 Bad Request` status.
    3. **Database Lookup for Webhook Registration**: It uses the extracted `externalResourceId` to query the `dbService` (`dbService.getWebhookRegistration(externalResourceId)`) and retrieve the corresponding `WebhookRegistryEntry`. This entry contains the `secret`, `externalWebhookId`, and the `registeredTasks` array. If no registration is found, it

logs a warning and returns `200 OK` (as it's a valid webhook, just not configured internally).

4. **Full Webhook Validation**: It then calls **`processWebhookRequest(webhookService, requestHeaders, webhookEntry.secret, rawRequest)`** again. This time, it passes the retrieved `secret` (or `externalWebhookId` for Google Drive) to perform **full signature/token validation** of the incoming webhook. If validation fails, it returns a `401 Unauthorized` status.
5. **Task Identification**: If the webhook is valid, it filters the initially identified tasks to include only those whose `id` is present in the `registeredTasks` array of the `WebhookRegistryEntry`. This ensures only tasks truly linked to *that specific external webhook registration* are triggered.
6. **Task Execution Loop**: It iterates through each identified task:
    - It fetches the latest `taskDefinition` from the database.
    - It retrieves the latest continuation tokens (`startPageToken`, `nextPageToken`) from the `WebhookRegistryEntry`.
    - It constructs a comprehensive `initialPayload` for the crawler, including the `taskDefinition`, processed `webhookPayload`, `externalResourceId`, `changeType`, and continuation tokens.
    - It then calls `await this.runOrchestrator(ctx, taskToExecute, initialPayload)` to initiate the actual crawl.
    - It captures the status of the first triggered task to return as the overall status.

- **Output**: Returns a `Promise<GSStatus>` indicating the overall success or failure of processing the webhook and triggering the associated tasks.

Future Scope

- **Asynchronous Task Dispatch**: For high-volume webhook events, instead of directly calling `runOrchestrator` in a loop, consider dispatching each task's `initialPayload` to an **asynchronous queue** (e.g., a message broker). This would prevent the webhook listener from being blocked by long-running crawls and improve responsiveness.
- **Concurrency Limits**: Implement logic to limit the number of concurrent executions for a single task or source to prevent resource exhaustion.
- **Error Reporting for Multiple Tasks**: If multiple tasks are triggered by one webhook and some fail, the current return status only reflects the `firstStatus`. Future improvements could aggregate results from all triggered tasks into a more comprehensive status report.
- **Webhook Payload Schema Validation**: Integrate more robust schema validation for incoming `rawRequest` payloads to catch malformed webhooks early.
- **Idempotency Handling**: Ensure that webhook events are processed idempotently, especially if the external service might send duplicate notifications. This might involve storing and checking a `message_id` from the webhook headers.
- **Retry Mechanisms**: Implement retry logic for failed `runOrchestrator` calls, possibly with exponential backoff.

- **Dead-Letter Queue (DLQ)**: For persistent failures, move problematic webhook events to a DLQ for manual inspection.

`triggerAllEnabledCronTasks` (Public Method)

TypeScript

```
// C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    /**
     * This method is designed to be called by a Godspeed cron event.
     * It checks all enabled cron tasks and triggers those that are due.
     * @param ctx The Godspeed context provided by the cron event.
     * @returns A GSStatus indicating the result of triggering tasks.
     */
    public async triggerAllEnabledCronTasks(ctx: GSContext): Promise<GSStatus> {
        logger.info("Manager received command to trigger all enabled cron tasks. Checking due tasks...");

        // Use ctx.event?.time for 'now' to align with Godspeed's event timestamp, with fallback
        const now = new Date((ctx as any).event?.time || new Date().toISOString());
        const results: { taskId: string; status: GSStatus }[] = [];
        let tasksDueCount = 0;

        const allTasks = await this.dbService.listAllTasks(); // Fetch all tasks from DB
        for (const task of allTasks) {
            logger.debug(`Checking task: ${task.id}, enabled: ${task.enabled}, trigger type: ${task.trigger.type}`);

            if (task.enabled && task.trigger.type === 'cron') {
                const cronTrigger = task.trigger as CronTrigger;
                try {
                    const interval = CronExpressionParser.parse(cronTrigger.expression, { currentDate: now });
                    const previousRunTime = interval.prev().toDate(); // Last scheduled time before or at 'now'

                    // Define a robust window (e.g., 65 seconds) to account for slight delays in cron execution
                    const sixtyFiveSecondsAgo = new Date(now.getTime() - (65 * 1000));
```

```
                // Condition:
                // 1. previousRunTime must be after the 'sixtyFiveSecondsAgo' mark (it's recent)
                // 2. previousRunTime must be at or before 'now' (it's not in the future)
                // 3. AND (crucially) the task's lastRun must be undefined (never run)
                //    OR the task's lastRun must be older than this specific previousRunTime.
                //    This prevents re-running a task for the same scheduled interval if the trigger fires
multiple times.
                if (previousRunTime > sixtyFiveSecondsAgo && previousRunTime <= now &&
                    (!task.lastRun || task.lastRun < previousRunTime)) {

                    logger.info(`Executing cron-triggered task: ${task.id} (expression:
${cronTrigger.expression}, last due: ${previousRunTime.toISOString()}).`);
                    tasksDueCount++;

                    // Enrich initialPayload with latest tokens from webhook registry if applicable
                    const sourceIdentifier = this.getSourceIdentifier(task.source.pluginType,
task.source.config);
                    let initialPayload: any = {};
                    if (sourceIdentifier) {
                        const webhookEntry = await
this.dbService.getWebhookRegistration(sourceIdentifier);
                        if (webhookEntry) {
                            initialPayload = {
                                startPageToken: webhookEntry.startPageToken,
                                nextPageToken: webhookEntry.nextPageToken,
                                otherCrawlerSpecificTokens: webhookEntry.otherCrawlerSpecificTokens
                            };
                        }
                    }
                    // Ensure taskDefinition is part of the initialPayload passed to runOrchestrator
                    initialPayload = {taskDefinition: task, ...initialPayload};
                    const status = await this.runOrchestrator(ctx, task, initialPayload);
                    results.push({ taskId: task.id, status });
                } else {
                    logger.debug(`Task '${task.id}' (cron: ${cronTrigger.expression}) not due. ` +
                                 `prevRun: ${previousRunTime.toISOString()}, ` +
                                 `now: ${now.toISOString()}, ` +
                                 `sixtyFiveSecondsAgo: ${sixtyFiveSecondsAgo.toISOString()}. ` +
                                 `lastRun: ${task.lastRun ? task.lastRun.toISOString() : 'never'}.`);
                }
            } catch (error: any) {
                logger.error(`Error parsing cron expression for task '${task.id}':
${cronTrigger.expression}. Error: ${error.message}`);
                results.push({ taskId: task.id, status: { success: false, code: 500, message: `Cron
expression parse error: ${error.message}` } });
```

```
            }
        }
    }

    if (tasksDueCount === 0) {
        logger.info("No enabled cron tasks were due at this time.");
        return new GSStatus(true, 200, "No enabled cron tasks were due.");
    }

    const successful = results.filter(r => r.status.success).length;
    const failed = results.length - successful;
    if (failed > 0) {
        return new GSStatus(false, 500, `Cron triggered ${tasksDueCount} tasks. ${successful}
succeeded, ${failed} failed.`, { data: results });
    }
    return new GSStatus(true, 200, `Successfully triggered ${successful} cron tasks.`, { data:
results });
  }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `triggerAllEnabledCronTasks` method is designed to be invoked by a **Godspeed cron event source**. Its purpose is to **identify and execute all enabled ingestion tasks** that are configured with a cron trigger and are currently due for a run. ⏰

Role / Logic

- **Input**: `ctx` (a `GSContext` object) which provides the current time of the cron event.
- **Process Flow**:
    1. **Log Initiation**: It logs a message indicating that it's checking for due cron tasks.
    2. **Get Current Time**: It determines the current time (`now`) from the `ctx.event.time` (or a fallback to `new Date()`).
    3. **Retrieve All Tasks**: It fetches all task definitions from the `dbService` (your in-memory database).
    4. **Iterate and Check Due Tasks**: For each `enabled` task with a `cron` trigger:
        - It uses `cron-parser` to calculate the `previousRunTime` (the last scheduled execution time that occurred at or before `now`).
        - It applies a **robust check** to ensure the task is genuinely due and hasn't been run for this specific interval already. This involves comparing `previousRunTime` against `now` and the task's `lastRun` timestamp, with a small buffer (65 seconds) to account for system delays.
        - If the task is due:

        - It logs the execution and increments `tasksDueCount`.
        - It retrieves the latest `taskDefinition` from the database (as a safeguard).
        - It enriches the `initialPayload` for the crawler with any relevant continuation tokens (`startPageToken`, `nextPageToken`) by looking up the associated `WebhookRegistryEntry` in the database, if a `sourceIdentifier` exists for the task.
        - It constructs the `initialPayload` to include the `taskDefinition` itself and any webhook-related data.
        - It then calls `await this.runOrchestrator(ctx, task, initialPayload)` to delegate the actual execution of the task.
        - The `status` of each triggered task is collected in the `results` array.
      - If the task is not due, it logs a debug message.
    5. **Handle Errors**: It includes a `try-catch` block to gracefully handle errors during cron expression parsing or task retrieval, logging them and marking the individual task's status as failed.
    6. **Summarize Results**: After checking all tasks, it counts successful and failed executions.
    7. **Return Overall Status**:
        - If `tasksDueCount` is 0, it returns a `GSStatus(true, 200)` indicating no tasks were due.
        - If there are `failed` tasks, it returns a `GSStatus(false, 500)` with a summary of successes and failures.
        - Otherwise, it returns a `GSStatus(true, 200)` for successful triggering of all due tasks.
- **Output**: Returns a `Promise<GSStatus>`.

Future Scope

- **Concurrency Control**: Implement mechanisms to prevent multiple concurrent cron triggers from initiating duplicate runs of the same task, especially for long-running tasks. This might involve a distributed lock.
- **Dynamic Cron Expression Updates**: If cron expressions change in the database, ensure the `cron-parser` re-evaluates them correctly without requiring a full application restart.
- **Metrics & Monitoring**: Enhance logging and integrate with a metrics system (e.g., Prometheus) to track the number of due tasks, execution times, and success/failure rates.
- **Scheduler Resilience**: For production, consider running this method in a highly available setup (e.g., multiple instances with a leader election mechanism) to ensure cron tasks are always triggered even if one scheduler instance fails.
- **Backoff/Retry for Failed Tasks**: If a task consistently fails, implement a backoff strategy to avoid overwhelming the external source or logging system.

runOrchestrator (Private Method)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleMana
ger.ts

// ... (previous code)

    private async runOrchestrator(ctx: GSContext, taskDefinition: IngestionTaskDefinition,
initialPayload?: any): Promise<GSStatus> {
        logger.info(`Running orchestrator for task '${taskDefinition.id}'`);
        this.eventBus.emit(IngestionEvents.TASK_TRIGGERED, taskDefinition.id);

        await this.dbService.updateTask(taskDefinition.id, { currentStatus:
IngestionTaskStatus.RUNNING });

        const sourceEntry = this.registeredPlugins.source.get(taskDefinition.source.pluginType);
        if (!sourceEntry) {
            const errorMessage = `Orchestrator failed: Source plugin
'${taskDefinition.source.pluginType}' not found for task '${taskDefinition.id}'`;
            logger.error(errorMessage);
            const status = new GSStatus(false, 500, errorMessage);
            this.eventBus.emit(IngestionEvents.TASK_FAILED, taskDefinition.id, status);
            await this.dbService.updateTask(taskDefinition.id, { currentStatus:
IngestionTaskStatus.FAILED, lastRunStatus: status });
            return status;
        }

        const destinationPlugin = taskDefinition.destination ?
this.registeredPlugins.destination.get(taskDefinition.destination.pluginType) : undefined;
        let destinationInstance: IDestinationPlugin | undefined;

        if (destinationPlugin && taskDefinition.destination) {
            destinationInstance = new destinationPlugin();
            await destinationInstance.init(taskDefinition.destination.config);
        }

        const sourceInstance = new sourceEntry.plugin({ config: taskDefinition.source.config });
        const orchestrator = new IngestionOrchestrator(
            sourceInstance,
            sourceEntry.transformer,
            destinationInstance,
```

```
            this.eventBus,
            taskDefinition.id
        );

        const finalStatus = await orchestrator.executeTask(ctx, initialPayload);

        if (finalStatus.success) {
            this.eventBus.emit(IngestionEvents.TASK_COMPLETED, taskDefinition.id, finalStatus);
            await this.dbService.updateTask(taskDefinition.id, { currentStatus:
IngestionTaskStatus.COMPLETED, lastRun: new Date(), lastRunStatus: finalStatus });
        } else {
            this.eventBus.emit(IngestionEvents.TASK_FAILED, taskDefinition.id, finalStatus);
            await this.dbService.updateTask(taskDefinition.id, { currentStatus:
IngestionTaskStatus.FAILED, lastRun: new Date(), lastRunStatus: finalStatus });
        }

        const sourceIdentifier = this.getSourceIdentifier(taskDefinition.source.pluginType,
taskDefinition.source.config);
        if (sourceIdentifier && finalStatus.data) {
            const updates: Partial<WebhookRegistryEntry> = {};
            if (finalStatus.data.startPageToken !== undefined) {
                updates.startPageToken = finalStatus.data.startPageToken;
                updates.nextPageToken = finalStatus.data.startPageToken; // Keep next and start in sync
for now for simplicity
            }
            if (finalStatus.data.nextPageToken !== undefined) {
                if (!updates.startPageToken) {
                    updates.nextPageToken = finalStatus.data.nextPageToken;
                }
            }
            if (finalStatus.data.otherCrawlerSpecificTokens !== undefined) {
                updates.otherCrawlerSpecificTokens = finalStatus.data.otherCrawlerSpecificTokens;
            }

            if (Object.keys(updates).length > 0) {
                try {
                    let webhookEntry = await this.dbService.getWebhookRegistration(sourceIdentifier);
                    if (!webhookEntry) {
                        webhookEntry = {
                            sourceIdentifier: sourceIdentifier,
                            endpointId: (taskDefinition.trigger as WebhookTrigger).endpointId || 'unknown',
                            secret: (taskDefinition.trigger as WebhookTrigger).secret || 'unknown',
                            externalWebhookId: (taskDefinition.trigger as WebhookTrigger).externalWebhookId
|| 'unknown',
                            registeredTasks: [taskDefinition.id],
                            webhookFlag: true
```

```
                    };
                    await this.dbService.saveWebhookRegistration(webhookEntry);
                    logger.warn(`GlobalIngestionLifecycleManager: Created missing webhook registry
entry for '${sourceIdentifier}' to save tokens.`);
                }
                await this.dbService.updateWebhookRegistration(sourceIdentifier, updates);
                logger.info(`GlobalIngestionLifecycleManager: Updated tokens for webhook registration
'${sourceIdentifier}'.`);
            } catch (dbError: any) {
                logger.error(`GlobalIngestionLifecycleManager: Failed to update webhook registration
tokens for '${sourceIdentifier}': ${dbError.message}`, { dbError });
            }
        }
    }
    return finalStatus;
  }
```

Brief Details

The `runOrchestrator` method is the **core execution engine** within the Scheduler SDK. It takes a task definition and an initial payload, then orchestrates the entire data ingestion pipeline for that single task run, from source data fetching to destination processing.

Role / Logic

- **Input**: `ctx` (Godspeed context), `taskDefinition` (the full `IngestionTaskDefinition`), and `initialPayload` (any data from the trigger, e.g., webhook body, continuation tokens).
- **Process Flow**:
    1. **Task Status Update**: Sets the `taskDefinition`'s `currentStatus` to `RUNNING` in the database.
    2. **Source Plugin Lookup**: Retrieves the `sourceEntry` (containing the `DataSource` class and `transformer`) from `this.registeredPlugins.source` based on `taskDefinition.source.pluginType`. If not found, it logs an error, updates the task status to `FAILED`, and returns.
    3. **Destination Plugin Initialization**: If a `destination` is defined in `taskDefinition`, it retrieves the `destinationPlugin` from `this.registeredPlugins.destination` and initializes a `destinationInstance`.
    4. **Crawler Instance Creation**: It **directly instantiates a new `sourceInstance`** of the specific `DataSource` class (e.g., `GitCrawlerDataSource`) using `taskDefinition.source.config`.
    5. **Orchestrator Invocation**: It creates a new `IngestionOrchestrator` instance, passing the `sourceInstance`, `transformer`, `destinationInstance`, `eventBus`, and `taskId`. It then calls `await orchestrator.executeTask(ctx, initialPayload)` to start the data flow.

6. **Result Handling & State Update**:
    - After `orchestrator.executeTask` completes, it updates the `taskDefinition`'s `currentStatus` (to `COMPLETED` or `FAILED`), `lastRun` timestamp, and `lastRunStatus` in the database.
    - If `finalStatus.success` and `finalStatus.data` contains `startPageToken`, `nextPageToken`, or `otherCrawlerSpecificTokens`, it extracts these and **persists them back into the associated `WebhookRegistryEntry`** in the database. This is vital for maintaining incremental sync state. If a `webhookEntry` doesn't exist for the `sourceIdentifier`, it logs a warning and creates a new entry before updating it.
7. **Event Emission**: Emits `TASK_TRIGGERED`, `TASK_COMPLETED`, or `TASK_FAILED` events on the `eventBus`.

- **Output**: Returns a `Promise<GSStatus>` reflecting the overall success or failure of the task execution.

Future Scope

- **Asynchronous Execution**: For long-running crawls, consider offloading the `orchestrator.executeTask` call to a background worker or message queue to prevent the `runOrchestrator` (and thus the triggering mechanism) from being blocked.
- **Concurrency Limits**: Implement a mechanism to limit the number of concurrent `runOrchestrator` executions for a given task or data source to prevent resource exhaustion.
- **Error Aggregation**: If multiple sub-steps within `orchestrator.executeTask` can fail, enhance the error reporting to provide more granular details in the `GSStatus` data.
- **Transactionality**: For real databases, ensure the `dbService.updateTask` and `dbService.updateWebhookRegistration` calls are part of a single transaction to maintain data consistency.
- **Dynamic Transformer/Destination Selection**: While `pluginType` is used, more advanced logic could allow for dynamic selection of transformers or destinations based on data characteristics or runtime conditions.

`onTaskCompleted` (Private Method)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    private onTaskCompleted(taskId: string, status: GSStatus) {
        logger.info(`Task '${taskId}' completed successfully.`);
```

```
  }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `onTaskCompleted` method is a **private event listener** within the `GlobalIngestionLifecycleManager`. It's triggered when an ingestion task successfully finishes its execution.

Role / Logic

- **Input**: `taskId` (string, the ID of the completed task) and `status` (`GSStatus`, the final status object from the task's execution).
- **Process**: It simply logs an informational message to the console, indicating that the specified task has completed successfully.
- **Output**: This method returns `void`.
- **Significance**: This method acts as a **callback** for the `TASK_COMPLETED` event emitted by the `IngestionOrchestrator` (or other parts of the Scheduler). While its current implementation is minimal (just logging), it serves as a crucial hook for future enhancements. The actual database update for task status is handled within `runOrchestrator`, so this listener primarily provides real-time feedback.

Future Scope

- **Notification System**: Integrate with external notification services (e.g., email, Slack, PagerDuty) to alert administrators about successful task completions.
- **Metrics Collection**: Push metrics (e.g., task duration, number of items processed) to a monitoring system (e.g., Prometheus, Datadog).
- **Downstream Workflow Triggering**: Trigger subsequent steps in a larger data pipeline (e.g., data validation, loading to a data warehouse, triggering an analytics job) by emitting new events or calling other services.
- **Audit Logging**: Write detailed audit logs about task completion for compliance or debugging.

`onTaskFailed` (Private Method)

```
TypeScript

//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)
```

```
    private onTaskFailed(taskId: string, status: GSStatus) {
        logger.error(`Task '${taskId}' failed.`);
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `onTaskFailed` method is a **private event listener** within the `GlobalIngestionLifecycleManager`. It's triggered when an ingestion task encounters an error and fails during its execution.

Role / Logic

- **Input**: `taskId` (string, the ID of the failed task) and `status` (`GSStatus`, the final status object indicating the failure).
- **Process**: It simply logs an error message to the console, indicating that the specified task has failed.
- **Output**: This method returns `void`.
- **Significance**: This method acts as a **callback** for the `TASK_FAILED` event emitted by the `IngestionOrchestrator` (or other parts of the Scheduler). Similar to `onTaskCompleted`, its current implementation is minimal, but it serves as a crucial hook for more advanced error management and notification systems. The actual database update for task status (to `FAILED`) is handled within `runOrchestrator`, so this listener primarily provides immediate feedback.

Future Scope

- **Alerting**: Integrate with alert systems (e.g., PagerDuty, email, Slack) to notify administrators of task failures, especially for critical tasks.
- **Error Reporting**: Send detailed error reports to an error tracking service (e.g., Sentry, Bugsnag) for deeper analysis.
- **Automated Retries**: Implement logic to automatically retry failed tasks, possibly with exponential backoff and a maximum number of retries.
- **Dead-Letter Queue (DLQ)**: For persistent or unrecoverable failures, move the task's payload or a summary to a DLQ for manual investigation.
- **Metrics Collection**: Push failure metrics (e.g., count of failed tasks, types of errors) to a monitoring system.

`getSourceIdentifier` (Private Method)

```
TypeScript
```

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleMana
ger.ts

// ... (previous code)

    private getSourceIdentifier(pluginType: string, sourceConfig: any): string | undefined {
        switch (pluginType) {
            case 'git-crawler':
                return sourceConfig.repoUrl;
            case 'googledrive-crawler':
                return sourceConfig.folderId;
            case 'teams-chat-crawler':
                return sourceConfig.meetingId || sourceConfig.chatId;
            case 'http-crawler':
                return sourceConfig.url;
            default:
                logger.warn(`Unsupported plugin type '${pluginType}' for source identification.`);
                return undefined;
        }
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `getSourceIdentifier` method is a **private helper function** used to extract a unique identifier for a data source from its configuration. This identifier is crucial for linking ingestion tasks to webhook registrations in the database.

Role / Logic

- **Input**: `pluginType` (string, e.g., `'git-crawler'`, `'googledrive-crawler'`) and `sourceConfig` (any, the configuration object specific to that data source).
- **Process**: It uses a `switch` statement to determine the `pluginType` and then returns the appropriate unique identifier based on the `sourceConfig`:
    - For `'git-crawler'`, it returns `sourceConfig.repoUrl`.
    - For `'googledrive-crawler'`, it returns `sourceConfig.folderId`.
    - For `'teams-chat-crawler'`, it returns `sourceConfig.meetingId` or `sourceConfig.chatId`.
    - For `'http-crawler'`, it returns `sourceConfig.url`.
    - If the `pluginType` is not recognized, it logs a warning and returns `undefined`.
- **Output**: Returns a `string` representing the unique source identifier, or `undefined` if the type is unsupported.

- **Significance**: This method centralizes the logic for deriving a consistent key to store and retrieve `WebhookRegistration` entries in the database, enabling the Scheduler SDK to manage webhooks for shared external resources.

Future Scope

- **Strict Type Checking**: Instead of `any` for `sourceConfig`, define specific type unions (e.g., `GitCrawlerConfig | GoogleDriveCrawlerConfig | HttpCrawlerConfig`) for `sourceConfig` to enable stricter type checking within the `switch` statement.
- **Error Handling for Missing Config**: If a required identifier (e.g., `repoUrl` for `git-crawler`) is missing from `sourceConfig`, consider throwing a more specific error instead of returning `undefined` and relying on a warning, especially if the `sourceIdentifier` is critical for subsequent operations.
- **Registry-Based Lookup**: For highly extensible systems, this logic could be part of a dynamic plugin registry where each plugin explicitly registers how its `sourceIdentifier` is derived.

`extractRepoNameFromUrl` (Private Method)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)

    private extractRepoNameFromUrl(repoUrl: string): string {
        try {
            const url = new URL(repoUrl);
            const pathParts = url.pathname.split('/').filter(part => part); // Remove empty strings
            if (url.hostname === 'github.com' && pathParts.length >= 2) {
                return `${pathParts[0]}/${pathParts[1]}`;
            }
        } catch (e: any) {
            logger.warn(`Failed to parse repo URL '${repoUrl}': ${e.message}`);
        }
        return repoUrl; // Fallback to original if not a GitHub URL or parsing fails
    }

// ... (rest of the GlobalIngestionLifecycleManager class)
```

Brief Details

The `extractRepoNameFromUrl` method is a **private helper function** designed to extract the "owner/repo" string from a full GitHub repository URL.

Role / Logic

- **Input**: `repoUrl` (string, the full URL of a GitHub repository, e.g., `https://github.com/owner/repo`).
- **Process**:
    1. It attempts to parse the `repoUrl` into a `URL` object.
    2. It checks if the hostname is `github.com` and if there are at least two path segments after the hostname (representing the owner and repository name).
    3. If both conditions are met, it constructs and returns the "owner/repo" string (e.g., `owner/repo`).
    4. If parsing fails, the hostname is not `github.com`, or the path structure is unexpected, it logs a warning and returns the original `repoUrl` as a fallback.
- **Output**: Returns a `string` in the format "owner/repo" for GitHub URLs, or the original URL if it's not a recognizable GitHub repository URL.
- **Significance**: This method is specifically used by the `registerWebhook` method for `git-crawler` to provide GitHub's API with the correct repository identifier format (e.g., `soham1334/Txt-to-Speech`) when registering webhooks, as GitHub's API often expects this format rather than the full URL.

Future Scope

- **Support for Other Git Providers**: Extend the logic to parse repository names from other Git hosting services (e.g., GitLab, Bitbucket) if the SDK is intended to support webhooks from those platforms.
- **Robustness**: Add more specific error handling or validation for malformed URLs beyond just logging a warning, especially if the returned `repoUrl` could cause issues downstream.
- **Centralized URL Parsing**: If similar URL parsing is needed elsewhere, consider extracting this into a more general utility.

`registerWebhook` (Public Method)

```
TypeScript


//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleManager.ts

// ... (previous code)
```

```
    public async registerWebhook(taskDefinition: IngestionTaskDefinition): Promise<GSStatus> {
        const trigger = taskDefinition.trigger as WebhookTrigger;
        const sourceConfig = taskDefinition.source.config;
        const pluginType = taskDefinition.source.pluginType;
        const taskId = taskDefinition.id;
        logger.debug("Entered into registerwebhook")
        const sourceIdentifier = this.getSourceIdentifier(pluginType, sourceConfig);
        if (!sourceIdentifier) {
            return new GSStatus(false, 400, `Webhook registration not supported or source identifier
missing for plugin type '${pluginType}'.`);
        }
        logger.debug(`---------sourceIdentifier:${sourceIdentifier}`)
        try {
            let existingWebhookEntry = await this.dbService.getWebhookRegistration(sourceIdentifier);
            let externalWebhookId: string;
            let secret: string;
            let channelResourceId: string | undefined;
            let registrationResultData: any = {};
            logger.debug(`----------------existingWebhookEntry:${existingWebhookEntry}`)
            if (existingWebhookEntry) {
                // Scenario 1: Webhook for this sourceIdentifier is already registered externally.
                // Do NOT re-register externally. Just ensure this task is linked.
                if (!existingWebhookEntry.registeredTasks.includes(taskId)) {
                    existingWebhookEntry.registeredTasks.push(taskId);
                    await this.dbService.updateWebhookRegistration(sourceIdentifier, { registeredTasks:
existingWebhookEntry.registeredTasks, webhookFlag: true });
                }
                externalWebhookId = existingWebhookEntry.externalWebhookId;
                secret = existingWebhookEntry.secret;
                channelResourceId = existingWebhookEntry.channelResourceId; // Retrieve existing
channelResourceId

                // Update task definition with existing webhook details
                trigger.externalWebhookId = externalWebhookId;
                trigger.secret = secret;
                trigger.channelResourceId = channelResourceId; // Update task with existing
channelResourceId

                await this.dbService.updateTask(taskId, { trigger: trigger });
                logger.info(`Task '${taskId}' associated with existing webhook for source
'${sourceIdentifier}'.`);
                return new GSStatus(true, 200, `Task associated with existing webhook for
'${sourceIdentifier}'.`);
            } else {
                // Scenario 2: Webhook for this sourceIdentifier is NOT yet registered externally.
```

```
            // Proceed with external registration.
            secret = crypto.randomBytes(20).toString('hex'); // Generate a new secret for this webhook
            let registrationStatus:any;
            logger.debug("------------proceeding to webhook registration------------")
            if (pluginType === 'git-crawler') {
                if (!trigger.credentials) {
                    throw new Error("Git crawler webhook registration requires credentials in trigger.");
                }
                const repoName = this.extractRepoNameFromUrl(sourceConfig.repoUrl);
                registrationStatus = await DataSourceApiUtils.registerWebhook(
                    pluginType,
                    trigger.credentials, // GitHub PAT
                    repoName,
                    trigger.endpointId,
                    secret
                );
            } else if (pluginType === 'googledrive-crawler') {
                if (!trigger.credentials) {
                    throw new Error("Google Drive crawler webhook registration requires credentials in
trigger.");
                }
                const serviceAccountCredentials = trigger.credentials as ServiceAccountKey; // Cast to
specific type
                if (!serviceAccountCredentials.serviceAccountKey &&
!serviceAccountCredentials.serviceAccountKeyPath) {
                    throw new Error("Google Drive crawler webhook registration requires service account
credentials (serviceAccountKey or serviceAccountKeyPath) in trigger credentials.");
                }
                registrationStatus = await DataSourceApiUtils.registerWebhook(
                    pluginType,
                    serviceAccountCredentials, // Service Account Key
                    sourceConfig.folderId,
                    trigger.endpointId,
                    secret
                );
            } else {
                return new GSStatus(false, 400, "Unsupported webhook plugin type for external
registration.");
            }
            logger.debug("DEBUG START --------registrationStatus from webhook reg----------")
            logger.debug(`registrationStatus: ${JSON.stringify(registrationStatus)}`)

            if (!registrationStatus.success || !registrationStatus.externalId) {
                throw new Error(`External webhook registration failed: ${registrationStatus.error ||
'Unknown error'}`);
            }
```

```
            logger.debug("DEBUG END----------------------------------")
            externalWebhookId = registrationStatus.externalId;
            channelResourceId = registrationStatus.channelResourceId; // Capture
channelResourceId from GDrive registration

            logger.debug(`---------- externalWebhookId:${ externalWebhookId}----------------`)
            registrationResultData = {
               startPageToken: registrationStatus.startpageToken,
               nextPageToken: registrationStatus.nextPageToken,
               otherCrawlerSpecificTokens: registrationStatus.otherCrawlerSpecificTokens
            };
            logger.debug(`-------------registrationResultData:${JSON.stringify(registrationResultData)}`)
            logger.info(`New webhook registered for '${sourceIdentifier}' with external ID
'${externalWebhookId}'. Channel Resource ID: '${channelResourceId || 'N/A'}'.`);

            const newWebhookEntry: WebhookRegistryEntry = {
               sourceIdentifier: sourceIdentifier,
               endpointId: trigger.endpointId,
               secret: secret,
               externalWebhookId: externalWebhookId,
               channelResourceId: channelResourceId, // Store channelResourceId
               registeredTasks: [taskId],
               webhookFlag: true,
               startPageToken: registrationResultData.startpageToken || undefined,
               nextPageToken: registrationResultData.nextPageToken || undefined,
               otherCrawlerSpecificTokens: registrationResultData.otherCrawlerSpecificTokens ||
undefined
            };
            logger.debug("-------------saving webhookEntry to db-----------")
            await this.dbService.saveWebhookRegistration(newWebhookEntry);

            // Update task definition with newly registered webhook details
            trigger.externalWebhookId = externalWebhookId;
            trigger.secret = secret;
            trigger.channelResourceId = channelResourceId; // Update task with new
channelResourceId

            // Update task's continuation tokens from registration response
            taskDefinition.startPageToken = newWebhookEntry.startPageToken;
            taskDefinition.nextPageToken = newWebhookEntry.nextPageToken;
            taskDefinition.otherCrawlerSpecificTokens =
newWebhookEntry.otherCrawlerSpecificTokens;
            logger.debug("-----------updating task-----------------")
            await this.dbService.updateTask(taskId, { trigger: trigger, ...newWebhookEntry });
```

```
            return new GSStatus(true, 200, `Webhook registered successfully for
'${sourceIdentifier}'.`);
        }
      } catch (error: any) {
        logger.error(`Failed to register webhook for task ${taskId} and source '${sourceIdentifier}':
${error.message}`, { error });
        return new GSStatus(false, 500, `Failed to register webhook: ${error.message}`);
      }
    }
```

Brief Details

The registerWebhook method is responsible for **managing external webhook subscriptions** for ingestion tasks. It handles both registering new webhooks with third-party services (like GitHub or Google Drive) and associating existing tasks with already registered webhooks.

Role / Logic

- **Input**: taskDefinition (an IngestionTaskDefinition object) which includes details about the webhook trigger and source configuration.
- **Process Flow**:
    1. **Source Identifier Check**: It first extracts the sourceIdentifier (e.g., repository URL, folder ID) from the taskDefinition. If this identifier is missing, it returns an error.
    2. **Check Existing Registration**: It queries the dbService (dbService.getWebhookRegistration(sourceIdentifier)) to check if a WebhookRegistration entry already exists for this sourceIdentifier.
        - **If an entry exists (Webhook Already Registered)**:
            - It logs that the task is being associated with an existing webhook.
            - It checks if the current taskId is already in the registeredTasks array of the existing webhookEntry. If not, it adds the taskId and updates the WebhookRegistration record in the database.
            - It retrieves the externalWebhookId, secret, and channelResourceId from the existing webhookEntry.
            - It then **updates the taskDefinition's trigger object** with these existing externalWebhookId, secret, and channelResourceId. This ensures the task has the correct IDs for future webhook validation.
            - It updates the task in the database and returns a success status.
        - **If no entry exists (Webhook Not Registered)**:
            - It generates a new unique secret for the webhook.
            - It determines the pluginType (git-crawler or googledrive-crawler) and prepares the necessary arguments for the external API call.

        - It uses **DataSourceApiUtils.registerWebhook()** (which dispatches to git-api-utils.ts or gdrive-api-utils.ts) to perform the actual registration with the external service.
            - For **GitHub**, it passes the repoName, callbackurl, secret, and credentials (PAT).
            - For **Google Drive**, it passes serviceAccountCredentials, folderId, callbackurl, and secret.
        - If external registration fails, it throws an error.
        - If successful, it captures the externalId (GitHub webhook ID or Google Channel ID), channelResourceId (for Google Drive), and any startPageToken/nextPageToken returned by the external service.
        - It creates a **new WebhookRegistryEntry** with these details and the current taskId in its registeredTasks array, then saves it to the database via dbService.saveWebhookRegistration().
        - Finally, it **updates the taskDefinition's trigger object** with the newly obtained externalWebhookId, secret, channelResourceId, and the startPageToken/nextPageToken for continuity. This updated taskDefinition is then persisted to the database.
    3. **Error Handling**: Catches any errors during the process (e.g., missing credentials, external API failures), logs them, and returns a GSStatus indicating failure.
- **Output**: Returns a Promise<GSStatus> indicating whether the webhook was successfully registered or associated with the task.

Future Scope

- **Retry Logic for External Registration**: Implement retry mechanisms with exponential backoff for DataSourceApiUtils.registerWebhook calls, as external API calls can be flaky.
- **Asynchronous External Registration**: For a production system, external webhook registration could be an asynchronous process (e.g., using a message queue) to prevent the scheduleTask method from blocking.
- **More Granular Error Codes**: Provide more specific GSStatus codes for different types of registration failures (e.g., 403 Forbidden for permission issues, 400 Bad Request for malformed requests).
- **Webhook Verification**: After successful registration, consider adding an optional step to verify the webhook's functionality by sending a test ping to the callbackurl and confirming a successful response.
- **Credential Management**: If credentials change, ensure the registerWebhook method can handle re-registering the webhook with the new credentials.
- **Handling channelResourceId for Git**: Currently channelResourceId is primarily for Google Drive. If Git webhooks also provide a similar resource ID, ensure it's captured and stored.

---

deregisterWebhook (Public Method)

TypeScript

```
//
C:\Users\SOHAM\Desktop\crawler\test-crawler\src\functions\ingestion\GlobalIngestionLifecycleMana
ger.ts

// ... (previous code)

    public async deregisterWebhook(taskId: string): Promise<GSStatus> {
        const task = await this.dbService.getTask(taskId);
        if (!task || task.trigger.type !== 'webhook') {
            return new GSStatus(false, 404, `Task with ID ${taskId} not found or is not a webhook task.`);
        }

        const trigger = task.trigger as WebhookTrigger;
        const sourceConfig = task.source.config;
        const pluginType = task.source.pluginType;

        const sourceIdentifier = this.getSourceIdentifier(pluginType, sourceConfig);
        if (!sourceIdentifier) {
            return new GSStatus(false, 400, `Webhook deregistration not supported or source identifier
missing for plugin type '${pluginType}'.`);
        }

        try {
            const webhookEntry = await this.dbService.getWebhookRegistration(sourceIdentifier);
            if (!webhookEntry) {
                logger.warn(`No webhook entry found for key '${sourceIdentifier}'. Assuming it's already
deregistered.`);
                return new GSStatus(true, 200, "Webhook already deregistered.");
            }

            const updatedRegisteredTasks = webhookEntry.registeredTasks.filter(id => id !== taskId);
            await this.dbService.updateWebhookRegistration(sourceIdentifier, { registeredTasks:
updatedRegisteredTasks });
            logger.info(`Task '${taskId}' removed from webhook registry for '${sourceIdentifier}'.
Remaining tasks: ${updatedRegisteredTasks.length}.`);

            if (updatedRegisteredTasks.length === 0) {
                logger.info(`Last task for webhook '${sourceIdentifier}' was removed. Deregistering
webhook externally.`);
                let deregistrationStatus: { success: boolean; message?: string };
```

```
            if (!webhookEntry.externalWebhookId) {
                logger.warn(`Cannot deregister external webhook for '${sourceIdentifier}':
externalWebhookId is missing from registry entry.`);
                return new GSStatus(false, 500, "Cannot deregister webhook: external ID missing.");
            }

            if (pluginType === 'git-crawler') {
                if (!trigger.credentials) {
                    throw new Error("Git crawler webhook deregistration requires credentials in trigger.");
                }
                const repoName = this.extractRepoNameFromUrl(sourceConfig.repoUrl);
                deregistrationStatus = await DataSourceApiUtils.deregisterWebhook(
                    pluginType,
                    repoName, // resourceId for Git is repoUrl
                    webhookEntry.externalWebhookId,
                    trigger.credentials,
                );
            } else if (pluginType === 'googledrive-crawler') {
                const serviceAccountCredentials = trigger.credentials as ServiceAccountKey;
                if (!serviceAccountCredentials.serviceAccountKey &&
!serviceAccountCredentials.serviceAccountKeyPath) {
                    throw new Error("Google Drive crawler webhook deregistration requires service
account credentials (serviceAccountKey or serviceAccountKeyPath) in trigger credentials.");
                }
                deregistrationStatus = await DataSourceApiUtils.deregisterWebhook(
                    pluginType,
                    sourceIdentifier, // resourceId for GDrive is folderId (sourceIdentifier)
                    webhookEntry.externalWebhookId,
                    serviceAccountCredentials,
                );
            } else {
                return new GSStatus(false, 400, "Unsupported webhook plugin type for external
deregistration.");
            }

            if (!deregistrationStatus.success) {
                throw new Error(deregistrationStatus.message);
            }

            await this.dbService.deleteWebhookRegistration(sourceIdentifier);
            logger.info(`External webhook '${webhookEntry.externalWebhookId}' for
'${sourceIdentifier}' deregistered.`);
            return new GSStatus(true, 200, "Webhook deregistered successfully.");
        }
```

```
            return new GSStatus(true, 200, "Task removed, but other tasks are still using this webhook.");
        } catch (error: any) {
            logger.error(`Failed to deregister webhook for task ${taskId} and source '${sourceIdentifier}': ${error.message}`, { error });
            return new GSStatus(false, 500, `Failed to deregister webhook: ${error.message}`);
        }
    }
```

Brief Details

The deregisterWebhook method handles the **removal of an ingestion task's association with an external webhook**. Crucially, it only performs the actual external API call to deregister the webhook (with services like GitHub or Google Drive) if the deleted task was the **last remaining task** linked to that specific webhook.

Role / Logic

- **Input**: taskId (string, the ID of the task being removed).
- **Process Flow**:
    1. **Task Validation**: It retrieves the task from dbService. If the task isn't found or isn't a webhook task, it returns a 404 Not Found status.
    2. **Source Identifier**: It determines the sourceIdentifier (e.g., repo URL, folder ID) for the task's source.
    3. **Retrieve Webhook Entry**: It fetches the webhookEntry from dbService.getWebhookRegistration(sourceIdentifier). If no entry is found (meaning the webhook is already gone or never existed), it logs a warning and returns 200 OK.
    4. **Update registeredTasks Array**: It filters the webhookEntry.registeredTasks array to remove the current taskId and updates this array in the database (dbService.updateWebhookRegistration).
    5. **Conditional External Deregistration**:
        - It checks if (updatedRegisteredTasks.length === 0). This is the **critical condition** for external deregistration.
        - If the array is empty, it proceeds to call DataSourceApiUtils.deregisterWebhook() (which dispatches to git-api-utils.ts or gdrive-api-utils.ts) to remove the webhook from the external service. It passes necessary details like pluginType, sourceIdentifier (as resourceId), webhookEntry.externalWebhookId, and trigger.credentials.
        - If the external deregistration fails, it throws an error.
        - If successful, it then deletes the WebhookRegistration record from the dbService.
        - It returns 200 OK for successful deregistration.

6. **Partial Deregistration**: If `updatedRegisteredTasks.length` is **not** zero, it means other tasks are still using this webhook. It logs this and returns `200 OK`, indicating the task was removed, but the external webhook remains active.
7. **Error Handling**: Catches any errors during the process, logs them, and returns a `500 Internal Server Error` status.

- **Output**: Returns a `Promise<GSStatus>`.

Future Scope

- **Asynchronous External Deregistration**: For production, offload the external API call to a background job or message queue to prevent the `deleteTask` method from blocking.
- **Retry Mechanisms**: Implement retry logic for `DataSourceApiUtils.deregisterWebhook` calls to handle transient network issues.
- **Forced Deregistration**: Add an option to force external deregistration even if `registeredTasks` is not empty (e.g., for cleanup of orphaned webhooks).
- **Audit Logging**: Log detailed information about successful and failed deregistration attempts, including external API responses.
