IngestionOrchestrator (Class)

TypeScript

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

import { IngestionData, IDestinationPlugin, IngestionDataTransformer, GSDataSource,
IngestionEvents } from './interfaces';
import { GSStatus, logger, GSContext } from '@godspeedsystems/core';
import { EventEmitter } from 'events';

export class IngestionOrchestrator extends EventEmitter {
  private sourceDataSource: GSDataSource;
  private dataTransformer: IngestionDataTransformer;
  private destination: IDestinationPlugin | undefined;
  private taskId: string;
  private eventBus: EventEmitter;

  constructor(
    source: GSDataSource,
    transformer: IngestionDataTransformer,
    destination: IDestinationPlugin | undefined,
    eventBus: EventEmitter,
    taskId: string
  ) {
    super();
    this.sourceDataSource = source;
    this.dataTransformer = transformer;
    this.destination = destination;
    this.eventBus = eventBus;
    this.taskId = taskId;
    logger.info(`IngestionOrchestrator instance created for task ${this.taskId}.`);
  }

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

Brief Details

The IngestionOrchestrator class is a **dedicated component responsible for managing the execution flow of a single ingestion task**. It acts as a coordinator, ensuring data moves sequentially from the source, through transformation, and to the destination.

Role / Logic

- **Inheritance**: It extends `EventEmitter`, allowing it to emit events related to the ingestion process (e.g., `DATA_FETCHED`, `DATA_TRANSFORMED`, `DATA_PROCESSED`).
- **Constructor Inputs**:
  - `source`: An instantiated `GSDataSource` (your crawler, e.g., `GitCrawlerDataSource`). This is the component responsible for fetching raw data.
  - `transformer`: An `IngestionDataTransformer` function. This function standardizes the raw data.
  - `destination`: An optional `IDestinationPlugin` instance. This component handles saving or sending the processed data to its final location.
  - `eventBus`: An `EventEmitter` instance, typically the one from `GlobalIngestionLifecycleManager`, used for emitting task-related events across the system.
  - `taskId`: The unique ID of the task this orchestrator instance is managing.
- **Initialization**: The constructor assigns these inputs to private instance properties and logs that an orchestrator instance has been created for the given `taskId`.

Future Scope

- **Error Handling Strategy**: While `executeTask` handles errors, the orchestrator could define a more explicit strategy for retries or error queues for sub-components.
- **Pipeline Configuration**: For more complex pipelines, the orchestrator could take a pipeline definition (e.g., an array of transformation steps) rather than just a single transformer.
- **Resource Management**: If crawlers or destinations require specific resource pools, the orchestrator could manage their allocation and release.

`getEventBus` (Public Method)

TypeScript

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

// ... (previous code)

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

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

Brief Details

The getEventBus method provides **access to the internal event emitter** specific to this IngestionOrchestrator instance.

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 the GlobalIngestionLifecycleManager (which creates this IngestionOrchestrator) to **subscribe to events** emitted by this particular orchestrator instance during its task execution (e.g., DATA_FETCHED, DATA_TRANSFORMED, DATA_PROCESSED). It's a key part of the communication back to the Scheduler.

Future Scope

- **Typed Events**: For enhanced type safety, consider using a more strongly typed event emitter library or pattern to define specific event names and their payload types, ensuring clarity on what data each event carries.
- **Restricted Access**: If the event bus should only be used internally, this method could be made private or removed, with events handled entirely within the IngestionOrchestrator and GlobalIngestionLifecycleManager directly.

executeTask (Public Method)

TypeScript

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

// ... (previous code)

    async executeTask(ctx: GSContext, initialPayload?: any): Promise<GSStatus> {
        if (!this.sourceDataSource || !this.dataTransformer) {
            const errorMessage = "Orchestrator not fully configured. DataSource and dataTransformer
are required.";
            logger.error(errorMessage);
            this.eventBus.emit(IngestionEvents.TASK_FAILED, this.taskId, { success: false, message:
errorMessage });
            return new GSStatus(false, 400, errorMessage);
        }

        logger.info(`Starting ingestion task execution for task ${this.taskId}...`);
        let totalItemsProcessed = 0;

        try {
```

```
    logger.info(`Orchestrator: Initializing Godspeed DataSource client
(${this.sourceDataSource.constructor.name}) for task ${this.taskId}...`);
    await this.sourceDataSource.initClient();
    logger.info(`Source client initialized for task ${this.taskId}.`);

    logger.info(`Orchestrator: Executing Godspeed DataSource
(${this.sourceDataSource.constructor.name}) with provided initialPayload...`);
    const sourceResultStatus: GSStatus = await this.sourceDataSource.execute(ctx,
initialPayload);

    let rawData: any[] = [];
    if (sourceResultStatus.success) {
      if (sourceResultStatus.data && sourceResultStatus.data.data) {
        rawData = Array.isArray(sourceResultStatus.data.data) ? sourceResultStatus.data.data
: [sourceResultStatus.data.data];
        logger.info(`Orchestrator: DataSource yielded ${rawData.length} data items from
'status.data.data'.`);
      } else if (sourceResultStatus.data) {
        rawData = [sourceResultStatus.data];
        logger.info(`Orchestrator: DataSource yielded 1 data item from 'status.data'.`);
      } else {
        logger.warn(`Orchestrator: Source executed successfully but returned no data in
'status.data' for task ${this.taskId}.`);
      }
    } else {
      const errorMessage = `Source execution failed for task ${this.taskId}:
${sourceResultStatus.message}`;
      logger.error(errorMessage, { data: sourceResultStatus.data });
      this.eventBus.emit(IngestionEvents.TASK_FAILED, this.taskId, { success: false, message:
errorMessage, data: sourceResultStatus.data });
      return new GSStatus(false, 500, errorMessage, { data: sourceResultStatus.data });
    }

    this.eventBus.emit(IngestionEvents.DATA_FETCHED, rawData, this.taskId);
    logger.info(`Orchestrator: Prepared ${rawData.length} raw data items for transformation.`);

    const payloadWithFetchedAt = { ...initialPayload, fetchedAt: new Date().toISOString() };
    logger.debug(`[Orchestrator DEBUG] Passing payload to transformer:`,
payloadWithFetchedAt);
    const transformedData: IngestionData[] = await this.dataTransformer(rawData,
payloadWithFetchedAt);

    this.eventBus.emit(IngestionEvents.DATA_TRANSFORMED, transformedData, this.taskId);
    logger.info(`Orchestrator: Transformed data, received ${transformedData.length} data
items.`);
```

```
      if (transformedData.length === 0) {
        logger.warn(`Orchestrator: No data ingested from source for task ${this.taskId}. Task completed with no data.`);
        const status = new GSStatus(true, 200, "Ingestion task completed: No data from source.", { itemsProcessed: 0 });
        this.eventBus.emit(IngestionEvents.TASK_COMPLETED, this.taskId, status);
        return status;
      }

      logger.info(`Orchestrator: Processing data for destination (if configured) for task ${this.taskId}...`);

      if (this.destination) {
        try {
          const sendResult = await this.destination.processData(transformedData);

          if (!sendResult.success) {
            logger.error(`Orchestrator: Destination processing failed for task ${this.taskId}: ${sendResult.message}`, { data: sendResult.data });
            const failureStatus = new GSStatus(false, 500, `Destination processing failed for task ${this.taskId}: ${sendResult.message}`, { itemsProcessed: totalItemsProcessed, data: sendResult.data });
            this.eventBus.emit(IngestionEvents.TASK_FAILED, this.taskId, failureStatus);
            return failureStatus;
          } else {
            totalItemsProcessed = transformedData.length;
            this.eventBus.emit(IngestionEvents.DATA_PROCESSED, transformedData, this.taskId);
            logger.info(`Orchestrator: Destination processing complete for task ${this.taskId}.`);
          }
        } catch (sendError: any) {
          logger.error(`Orchestrator: Error during destination processing for task ${this.taskId}: ${sendError.message}`, { error: sendError });
          const failureStatus = new GSStatus(false, 500, `Error during destination processing for task ${this.taskId}: ${sendError.message}`, { itemsProcessed: totalItemsProcessed, data: sendError.message });
          this.eventBus.emit(IngestionEvents.TASK_FAILED, this.taskId, failureStatus);
          return failureStatus;
        }
      } else {
        totalItemsProcessed = transformedData.length;
        logger.info(`Orchestrator: No destination configured for task ${this.taskId}. Data considered processed after transformation.`);
      }
```

```
            logger.info(`Ingestion task ${this.taskId} completed. Total items processed/emitted:
${totalItemsProcessed}.`);
            const successStatus = new GSStatus(true, 200, "Ingestion task completed successfully.", {
itemsProcessed: totalItemsProcessed });
            this.eventBus.emit(IngestionEvents.TASK_COMPLETED, this.taskId, successStatus);
            return successStatus;

        } catch (error: any) {
            const errorMessage = `Ingestion task ${this.taskId} failed: ${error.message}`;
            logger.error(errorMessage, { error: error });
            const failureStatus = new GSStatus(false, 500, errorMessage, { itemsProcessed:
totalItemsProcessed, data: error.message });
            this.eventBus.emit(IngestionEvents.TASK_FAILED, this.taskId, failureStatus);
            return failureStatus;
        }
    }
```

Brief Details

The `executeTask` method is the **central orchestrator for a single ingestion task run**. It coordinates the entire pipeline: initializing the data source, fetching raw data, transforming it, and sending it to a destination, while emitting events at each stage.

Role / Logic

- **Input**:
    1. `ctx`: The Godspeed context (`GSContext`).
    2. `initialPayload`: An optional object containing data from the task's trigger (e.g., webhook payload, continuation tokens).
- **Process Flow**:
    1. **Configuration Check**: Verifies that `sourceDataSource` and `dataTransformer` are properly configured. If not, it logs an error, emits `TASK_FAILED`, and returns a `400 Bad Request`.
    2. **Source Initialization**: Calls `this.sourceDataSource.initClient()` to ensure the crawler's client is ready.
    3. **Data Extraction**: Calls `this.sourceDataSource.execute(ctx, initialPayload)` to fetch raw data from the external source.
    4. **Raw Data Handling**:
        - If `sourceDataSource.execute` is successful, it extracts `rawData` from the returned `GSStatus`.
        - If `sourceDataSource.execute` fails, it logs the error, emits `TASK_FAILED`, and returns a `500 Internal Server Error`.
    5. **Data Transformation**: Emits `DATA_FETCHED`, then calls `this.dataTransformer(rawData, payloadWithFetchedAt)` to transform the raw data

into IngestionData. It adds a fetchedAt timestamp to the payload passed to the transformer.

6. **Transformed Data Check**: If transformedData is empty, it logs a warning, emits TASK_COMPLETED, and returns a successful GSStatus (as there's no data to process further).
7. **Destination Processing**:
   - If a this.destination plugin is configured, it calls this.destination.processData(transformedData).
   - If processData fails, it logs an error, emits TASK_FAILED, and returns a 500 Internal Server Error.
   - If processData succeeds, it emits DATA_PROCESSED.
   - If no destination is configured, it logs that data is considered processed after transformation.
8. **Final Status & Event Emission**: Logs the completion, emits TASK_COMPLETED, and returns a successful GSStatus with the total number of items processed.
9. **Error Catch-all**: A try-catch block wraps the entire execution, catching any unexpected errors, logging them, emitting TASK_FAILED, and returning a 500 Internal Server Error.

- **Output**: Returns a Promise<GSStatus> indicating the overall outcome of the task execution.

Future Scope

- **Batch Processing**: For very large datasets, implement internal batching mechanisms to process data in smaller chunks, reducing memory footprint.
- **Retry Logic**: Add retry logic for sourceDataSource.execute and destination.processData calls to handle transient failures.
- **Circuit Breaker**: Implement a circuit breaker pattern for external API calls within the data source and destination to prevent cascading failures.
- **Metrics & Tracing**: Enhance logging with more detailed metrics and integrate with distributed tracing systems for better observability of the entire pipeline.
- **Dynamic Pipeline**: Allow the dataTransformer and destination to be part of a dynamically configurable pipeline (e.g., an array of transformers) rather than single instances.
- **Data Validation**: Implement stricter validation after data fetching and transformation to ensure data quality before sending to the destination.