

Design Document for Distributor Data Extraction Ingestion

Nathan Lunceford

2025-02-14

Table of contents

1	Overview	2
2	System Architecture	2
2.1	Entry Point and Application Hosting	2
2.2	Extraction Ingestion Orchestration	3
2.3	External Dependencies	3
3	Design Patterns Employed	3
3.1	Dependency Injection (DI)	3
3.2	Registry Pattern	4
3.3	Abstract Factory Pattern	4
3.4	Builder Pattern	4
3.5	Strategy Pattern	5
4	Detailed Component Descriptions	5
4.1	Program Class	5
4.2	ERPService Class	6
4.3	Providers	6
4.4	Data Loader – S3DataLoader	7
4.5	Registries	7
4.6	Builder Components	7
4.6.1	APIRequestBuilder & AuthenticationBuilder	7
4.6.2	DatabaseQueryBuilder	7
4.7	Configuration Models	8
5	Pseudo-code for Nomad Integration	8

6 Error Handling and Logging	11
7 External Dependencies and Integration	11
8 Conclusion	12

1 Overview

This design document covers the extraction ingestion component of our ETL pipeline specifically for distributor data. In our architecture, distributors integrate with our system via two primary methods:

- **Direct Integration:** Where our system automatically provisions the correct resources based solely on the three supplied parameters—**erp-type**, **client-id**, and **data-type**.
- **SFTP Ingestion:** Where distributors SFTP their data to our environment (this method is handled separately).

The design detailed here pertains only to the direct extraction ingestion method. This process extracts distributor data and deposits it into a pre-dropzone S3 bucket. At this stage, a minor transformation is applied: column names are standardized according to our global data dictionary and the output files are converted to Parquet format. More complex transformations occur later in the pipeline.

2 System Architecture

2.1 Entry Point and Application Hosting

- **Program Class:**
 - **Main Method:**
 - * Parses command-line options (e.g., `--erp-type`, `--client-id`, `--data-type`) using `System.CommandLine`.
 - * Sets up the dependency injection container and configures AWS services.
 - * Resolves the primary extraction ingestion service (`ERPService`) from the DI container and triggers the extraction process.
 - **CreateHostBuilder Method:**
 - * Configures the host with AWS services, core providers, registries, and builder components.

2.2 Extraction Ingestion Orchestration

- **ERPService Class:**
 - Acts as the orchestrator for the extraction ingestion process.
 - Retrieves credentials and configuration from AWS Secrets Manager and DynamoDB, respectively.
 - Dynamically resolves components for data extraction based on ERP type.
 - Supports two extraction modes:
 - * **API Mode:** For ERPs that expose APIs.
 - * **Database Mode:** For ERPs that require direct database access.
 - Applies a minimal transform:
 - * Standardizes column names according to our global data dictionary.
 - * Converts all files to Parquet format.
 - Deposits the extracted (and minimally transformed) data into a pre-dropzone S3 bucket.
 - (Note: Subsequent, more complex transformations are performed later in the overall ETL pipeline.)

2.3 External Dependencies

- **AWS Services:**
 - **IAmazonSecretsManager:** Retrieves ERP credentials.
 - **IAmazonDynamoDB:** Fetches ERP configuration data.
 - **IAmazonS3:** Loads the extracted data into the pre-dropzone S3 bucket.
- **.NET Libraries:**
 - **System.CommandLine:** For command-line parsing.
 - **Microsoft.Extensions.Hosting & DI:** For hosting and dependency injection.
 - **System.Text.Json:** For JSON serialization/deserialization.

3 Design Patterns Employed

3.1 Dependency Injection (DI)

- **Usage:**
 - Decouples service construction from business logic.
 - Registers AWS clients, providers, registries, builder components, and the main extraction service in the DI container.

- **Benefits:**
 - Enhances testability and maintainability.
 - Promotes separation of concerns.

3.2 Registry Pattern

- **Components:**
 - ERPRegistry, ExtractorRegistry, TransformationRegistry, and LoaderRegistry.
- **Usage:**
 - Centralizes lookup for ERP-specific factories and strategies.
 - Dynamically resolves connectors, extractors, transformers, and loaders based on ERP type or data type.
- **Benefits:**
 - Simplifies addition of new ERP integrations.
 - Reduces direct dependencies between the extraction process and concrete implementations.

3.3 Abstract Factory Pattern

- **Component:**
 - The IERPFactory interface (managed via ERPRegistry).
- **Usage:**
 - Encapsulates creation of ERP-specific components (e.g., connectors and jobs).
- **Benefits:**
 - Supports multiple ERP systems with varying implementations without altering extraction logic.

3.4 Builder Pattern

- **Components:**
 - **APIRequestBuilder & AuthenticationBuilder:**
 - * Provide fluent interfaces to construct complex API request objects.
 - **DatabaseQueryBuilder (New):**
 - * Dynamically constructs SQL queries for ERPs requiring direct database access.

- **Usage:**
 - The `DatabaseQueryBuilder` collects parameters (e.g., ERP type, connection string, schema, table, etc.) and produces a `DatabaseQuery` object with a `GenerateSql` method.
- **Benefits:**
 - Enhances readability and modularity.
 - Supports both API and database extraction modes seamlessly.

3.5 Strategy Pattern

- **Components:**
 - Interfaces such as `IExtractor` and `ITransformer`.
- **Usage:**
 - Encapsulate different implementations for data extraction and minimal transformation.
 - Registries (like `ExtractorRegistry` and `TransformationRegistry`) select the appropriate strategy at runtime.
- **Benefits:**
 - Provides flexibility to extend or change extraction and transformation algorithms without impacting the overall system.

4 Detailed Component Descriptions

4.1 Program Class

- **Main Method:**
 - Parses Nomad-supplied command-line arguments (only **erp-type**, **client-id**, and **data-type** are required).
 - Builds the DI container via `CreateHostBuilder`.
 - Resolves and invokes `ERPService.ProcessERPData`.
 - Handles global errors for graceful failure.
- **CreateHostBuilder Method:**
 - Configures services including AWS clients, providers, registries, and builder components (for both API and database queries).

4.2 ERPService Class

- **Responsibilities:**
 - Orchestrates the extraction ingestion process.
 - Retrieves credentials and ERP configuration.
 - Dynamically resolves ERP-specific components using registries.
 - Chooses between API or Database extraction modes based on the ERP configuration.
 - Applies a minor transformation to standardize column names (per the global data dictionary) and converts files to Parquet.
 - Deposits the minimally transformed data into a pre-dropzone S3 bucket.
- **Key Method – ProcessERPData:**
 - **Credential & Configuration Retrieval:**
 - * Uses `ICredentialProvider` and `IConfigurationProvider` to obtain ERP settings.
 - **Dynamic Component Resolution:**
 - * Uses registries to resolve ERP-specific factories, extractors, transformers, and loaders.
 - **Integration Modes:**
 - * **API Mode:**
 - Builds an API request via `APIRequestBuilder` (and `AuthenticationBuilder`) and extracts data via `IExtractor.Extract`.
 - * **Database Mode:**
 - Builds a SQL query using `DatabaseQueryBuilder` and extracts data via `IExtractor.ExtractFromDatabase`.
 - **Subsequent Steps:**
 - * Applies the minor transform (standardizes column names and converts to Parquet).
 - * Loads the resulting data into a pre-dropzone S3 bucket via `IDataLoader`.

4.3 Providers

- **AWSCredentialProvider:**
 - Retrieves and deserializes credentials from AWS Secrets Manager.
- **DynamoDBConfigProvider:**
 - Fetches ERP configuration from DynamoDB and maps it to an `ERPConfiguration` object.

- **New Configuration Fields:**
 - * **AccessType:** Indicates if the ERP uses API or Database.
 - * **ConnectionString, Schema, and BatchSize** for database integrations.

4.4 Data Loader – S3DataLoader

- **Responsibilities:**
 - Formats and loads the minimally transformed data into S3.
 - Converts files to Parquet format and ensures standardized column names.
- **Key Methods:**
 - **Load:** Manages the upload process.
 - **FormatData:** Applies the transformation (e.g., renaming columns per the global data dictionary and converting to Parquet).

4.5 Registries

- **LoaderRegistry, ExtractorRegistry, TransformationRegistry, ERPRegistry:**
 - Maintain mappings from ERP type or data type to concrete implementations.
 - Provide lookup methods (e.g., `GetLoader`, `GetExtractor`, `GetStrategy`, `GetFactory`) for dynamic resolution.

4.6 Builder Components

4.6.1 APIRequestBuilder & AuthenticationBuilder

- **Usage:**
 - Allow fluent construction of API requests.
 - Support chaining methods to specify ERP type, endpoint, HTTP method, authentication, headers, query parameters, retry policy, and timeout.

4.6.2 DatabaseQueryBuilder

- **Responsibilities:**
 - Provides a fluent interface for building SQL queries for ERP systems that require direct database access.
- **Chainable Methods:**

- ForERP, WithConnectionString, WithSchema, WithTable, WithColumns, WithWhere, WithOrderBy, WithLimit, WithOffset, WithParameter, WithCommandTimeout, WithIsolationLevel
- **Build Method:**
 - Validates required parameters and constructs a `DatabaseQuery` object.
- **DatabaseQuery Object:**
 - Contains properties for ERP type, connection string, schema, table, columns, conditions, ordering, limits, parameters, command timeout, and isolation level.
 - Provides a `GenerateSql` method to convert query parameters into a valid SQL string.

4.7 Configuration Models

- **ERPConfiguration:**
 - Stores settings such as `BaseUrl`, `CompanyId`, `WarehouseId`, `RequiredHeaders`, and timeout/retry settings.
 - **New Fields for Database Access:**
 - * `AccessType`: Enum (API or Database).
 - * `ConnectionString`: For direct database connections.
 - * `Schema`: Database schema.
 - * `BatchSize`: Number of records to fetch per batch.
- **ERPCredentials:**
 - Holds secure API keys and client secrets.
- **LoadConfiguration:**
 - Used by the data loader to configure the S3 upload.
- **AccessType Enum:**
 - Distinguishes between API and Database access modes.

5 Pseudo-code for Nomad Integration

```
Main:
// Parse Nomad-supplied command-line arguments:
// --erp-type, --client-id, --data-type
```



```

options = parseArguments(["--erp-type", "--client-id", "--data-type"])

// Build the host container with dependency injection configured
host = createHostBuilder().build()

// Retrieve the ERPService from the DI container
erpService = host.getService(ERPService)

// Trigger the extraction ingestion process with the supplied parameters
erpService.ProcessERPData(options.erpType, options.clientId, options.dataType)

ERPService.ProcessERPData(erpType, clientId, dataType):
    Log "Starting ETL extraction ingestion for client [clientId] using ERP [erpType]"

    // Retrieve ERP credentials (from AWS Secrets Manager)
    credentials = CredentialProvider.GetCredentials(erpType, clientId)

    // Retrieve ERP configuration (from DynamoDB)
    erpConfig = ConfigurationProvider.GetConfiguration(erpType, clientId)

    // Lookup common components via registries:
    //   - ERP-specific factory (for connectors and jobs)
    //   - Data extractor (for API or DB extraction)
    //   - Data transformer (to standardize columns and convert to Parquet)
    //   - Data loader (to upload data to the pre-dropzone S3 bucket)
    factory = ERPRegistry.GetFactory(erpType, clientId)
    extractor = ExtractorRegistry.GetExtractor(erpType)
    transformer = TransformationRegistry.GetStrategy(erpType, dataType)
    loader = LoaderRegistry.GetLoader("s3")

    if erpConfig.AccessType == Database then:
        // DATABASE MODE: Build SQL query using the DatabaseQueryBuilder
        query = DatabaseQueryBuilder()
            .ForERP(erpType)
            .WithConnectionString(erpConfig.ConnectionString)
            .WithSchema(erpConfig.Schema)
            .WithTable(dataType + "_table")
            .WithColumns("id", "created_at", "data")
            .WithWhere("is_processed", false)
            .WithOrderBy("created_at")
            .WithLimit(erpConfig.BatchSize)

```

```

        .WithCommandTimeout(erpConfig.TimeoutSeconds)
        .Build()

    Log "Executing database query: " + query.GenerateSql()
    // Extract data directly from the database
    extractedData = extractor.ExtractFromDatabase(query)
else:
    // API MODE: Build an API request using the APIRequestBuilder and AuthenticationBuilder
    request = APIRequestBuilder()
        .ForERP(erpType)
        .WithEndpoint(erpConfig.BaseUrl + "/api/v2/sales")
        .WithMethod(GET)
        .WithAuthentication(
            AuthenticationBuilder()
                .WithApiKey(credentials.ApiKey)
                .WithClientId(credentials.ClientId)
                .WithClientSecret(credentials.ClientSecret)
                .Build()
        )
        .WithHeaders(erpConfig.RequiredHeaders)
        .WithQueryParameters({
            "companyId": erpConfig.CompanyId,
            "warehouse": erpConfig.WarehouseId,
            "pageSize": erpConfig.PageSize.toString()
        })
        .WithRetryPolicy(erpConfig.MaxRetries)
        .WithTimeout(erpConfig.TimeoutSeconds)
        .Build()

    Log "Executing API request to " + erpConfig.BaseUrl + "/api/v2/sales"
    // Extract data via the API
    extractedData = extractor.Extract(request)

// Create ERP-specific components using the abstract factory pattern
connector = factory.CreateConnector()
job = factory.CreateJob()

// Transform the extracted data (standardize columns and convert to Parquet)
Log "Starting minor data transformation"
transformedData = transformer.Transform(extractedData)

// Configure the load operation for the pre-dropzone S3 bucket

```

```

loadConfig = new LoadConfiguration(
    Bucket: "erp-data-" + clientId,
    Key: erpType + "/" + dataType + "/" + currentTimestamp + "/data.parquet",
    Format: Parquet,
    Metadata: {
        "erp_type": erpType,
        "client_id": clientId,
        "data_type": dataType,
        "extract_timestamp": currentTimestamp
    }
)

// Load the transformed data into S3
Log "Starting data load to S3 pre-dropzone"
loader.Load(transformedData, loadConfig)

Log "Extraction ingestion process completed successfully"

```

6 Error Handling and Logging

- **Error Handling:**
 - Try-catch blocks around critical operations (AWS calls, extraction, and query building).
 - Validations in builder components ensure required fields are provided.
- **Logging:**
 - `ILogger<T>` is used to log key events and errors.
 - Detailed logs enable tracing of the extraction ingestion workflow.

7 External Dependencies and Integration

- **AWS Services:**
 - **Secrets Manager:** Securely retrieves credentials.
 - **DynamoDB:** Provides ERP configuration data.
 - **S3:** Stores the minimally transformed data in the pre-dropzone.
- **Database Integration:**

- For ERP systems without APIs, direct database queries are supported using `DatabaseQueryBuilder` and `DatabaseQuery`.
- **.NET Libraries:**
 - **System.CommandLine:** For CLI parsing.
 - **Microsoft.Extensions.Hosting/DI:** For application hosting and dependency injection.
 - **JsonSerializer:** For data serialization tasks.

8 Conclusion

The extraction ingestion component of the ETL pipeline for distributor data automatically provisions the correct resources based solely on three Nomad-supplied parameters (**erp-type**, **client-id**, and **data-type**). It supports both API and database extraction modes and applies a minor transformation—standardizing column names and converting files to Parquet—before depositing the data into a pre-dropzone S3 bucket. More complex transformations occur later in the overall pipeline, and a separate ingestion method is provided for distributors that SFTP their data. The architecture and design patterns employed enable a flexible, maintainable, and scalable solution.