Link Event Catalog

This is the central documentation portal for all domains, services, and events in the Link platform. Use this catalog to discover existing contracts, understand dependencies, and explore our event-driven architecture.

Generated on 4/28/2026 3:53:46 PM

Table of Contents

Domains 5
- Tenant 6
- Report 8
- Knowledge Artifact Management 10
- Data Access 12
- Compliance 14

Services 15
- Admin UI (front-end) 16
- Admin Backend-for-Frontend 20
- Tenant Service 21
- Census Service 24
- Submission Service 37
- Report Service 45
- Report Service 50
- Query Dispatch Service 55
- Normalization Service 56
- Measure Evaluation Service 59
- Measure Evaluation Service 63
- Terminology Service 67
- Data Acquisition Worker Service 69
- Data Acquisition Service 71
- Validation Service 79
- Audit Service 88
- Account Service 89

Events 90
- Validation Complete 91
- Resource Normalized 92
- Resource Normalized 93
- Resource Evaluated 94
- Resource Acquired 95
- Report Status Updated 96
- Report Scheduled 97
- Ready to Acquire 98
- Ready for Validation 99
- Payload Submitted 100
- Patient Lists Acquired 101
- Patient Event 102
- Patient Census Scheduled 103
- Measure Report Generated 104
- Measure Evaluated 105

Auditable Event Occurred 106
Commands 107
Validate ValueSet Code 108
Validate CodeSystem Code 110
Submit Report 112
Submit Report 113
Generate Report Requested 114
Evaluation Requested 115
Data Acquisition Requested 116
Queries 117
Get Service Info 118
Get Facility Configuration 119
Get Facilities List 120
Extra/Custom Documentation 121
Development Docs 122
Testing 123
Tail Messages 131
Open Telemetry 135
Logging & Error Handling 142
Development Patterns 149
Kafka DeadLetter/Retry Scenarios 150
Authorization Policies 157
Application Configuration Management Process 160
API Guidance 162
Design Docs 173
NHSN Reporting Use Case 174
Telemetry 177
Security 179
Retry Topics 182
Performance Model 185
Architecture References 193
Dual Scheduler Architecture 194
Data Persistence 206
Auth Flow 210
Deployment & Usage 214
Tenant Management 215
Link Product Overview 218
Configuring Java Services 220
Deployment Configuration 227
Configuring .NET Services 228

Domains

Tenant

Type: Domain

Overview

Tenant abstracts entities with specific authority and access to Link's behaviors and information.

Examples:

- A hospital system using Link to report performance of one of its hospitals to a federal agency is a tenant.
- A report consumer that receives access to an individual report, or even a reporting stream, is a tenant.
- The systems operator with deployment-wide authority to all information and behavior in Link is a tenant.

Requirements

- As a tenant, I want to configure so that my Link requirements are satisfied
  - As a tenant, I want to configure reporting so that my reporting requirements are satisfied
    - As a tenant, I want to plan reports so that my reporting requirements are satisfied
      - As a tenant, I want to plan reports so that my reporting requirements are satisfied
      - AC: Able to set Cron-based recurrence
      - AC: Able to

  - As a tenant, I want to configure data sources so that my reporting requirements are satisfied
  - As a tenant, I want to configure data transformation so that my reporting requirements are satisfied
  - As a tenant, I want to configure reporting organizations so that my reporting requirements are satisfied

Diagrams

Flow Chart

```
flowchart TD
 subgraph subGraph0["Link"]
        Tenant["«Domain» <br> Tenant"]
        Report["«Domain» <br> Report"]
        DataAccess["«Domain» <br> Data Access"]
        Compliance["«Domain» <br> Compliance"]
        DataSource["Data Source"]
        ReportConsumer["Report Consumer"]
        Org["Reporting Organization"]
  end
  subgraph subGraph1["Knowledge Artifact Repository"]
    dQM
```

```
    IG
    Profile
    Terminology
  end
    Tenant -- plans --> Report
    Tenant -- configures --> DataAccess
    Report -- uses data from --> DataAccess
    DataAccess -- acquires from --> DataSource
    Org -- reports --> Report
    Report -- consumed by --> ReportConsumer
    Report -- based on --> dQM["dQM"]
    Compliance -- assures --> Report
    Tenant -- has a set of--> Org
    style Tenant fill:#dff,stroke:#9ff,stroke-width:2px
    style Report fill:#dff,stroke:#9ff,stroke-width:2px
    style DataAccess fill:#dff,stroke:#9ff,stroke-width:2px
    style Compliance fill:#dff,stroke:#9ff,stroke-width:2px
```

Report

Type: Domain

Overview

The Report domain encompasses the full lifecycle of data extraction, transformation, evaluation, and delivery for reporting workflows. It orchestrates the flow of raw clinical data into structured, validated, and submitted quality reports. Each service plays a specialized role in enabling modularity, traceability, and performance at scale.

Definitions

Concept	Description
Report	The output of Link's evaluation of data. A report can be submitted to NHSN, it can be evaluated in Power BI, it can be visualized in a custom UI.
Evaluation	The process of doing something with data within Link (i.e. acquire, normalize, check criteria, submit).
Reporting Requirement	A description of the need for data in a report. (i.e. "Bed capacity reporting during a pandemic" or "Hypoglycemic patients")
Reporting Orchestration	Indicates the steps to take within the Link system to produce reportable data.

Stakeholder Requirements

- Reporting functionality SHALL be modular so that it can be used for multiple use cases
- Link acquires (either via PUSH or PULL) and evaluates the data in a reporting pipeline.
- Reporting pipelines SHALL be orchestrated via configuration of services
- As a reporting organization and reporting consumer, I want the system to retain version history of IGs used during reporting in a reporting period, so that deleted reports can be accurately regenerated using their original definitions.

Functional Requirements

TODO

Diagrams

State

```
stateDiagram-v2
state "Report Scheduled" as ReportScheduled
state "Data Acquisition Requested" as DataAcquisitionRequested
state "Patient Lists Acquired" as PatientListsAcquired
state "Resource Evaluated" as ResourceEvaluated
state "Validation Complete" as ValidationComplete
state "Report Submitted" as ReportSubmitted
state "Report Generation Failed" as ReportGenerationFailed
state "Submission Failed" as SubmissionFailed

ReportScheduled --> DataAcquisitionRequested : GenerateReportValue
DataAcquisitionRequested --> PatientListsAcquired : PatientListsAcquiredValue
PatientListsAcquired --> ResourceEvaluated : ResourceEvaluatedValue (for each patient)
ResourceEvaluated --> ValidationComplete : ReadyForValidationValue
ValidationComplete --> ReportSubmitted : ReportSubmittedValue

DataAcquisitionRequested --> ReportGenerationFailed : Error
PatientListsAcquired --> ReportGenerationFailed : Error
ResourceEvaluated --> ReportGenerationFailed : Error
ValidationComplete --> ReportGenerationFailed : Error
ReportSubmitted --> SubmissionFailed : Error

ReportGenerationFailed --> ReportScheduled : Retry (GenerateReport)
SubmissionFailed --> ReportSubmitted : Retry (SubmitReport)

state "Initial" as Initial
Initial --> ReportScheduled : Report Scheduled
```

Knowledge Artifact Management

Type: Domain

Overview

The Knowledge Artifact Management domain represents the organizational and governance processes that support and surround the technical domains within the Link ecosystem.

This domain includes key artifacts such as:

- FHIR Implementation Guides (IGs), including FHIR Profiles
- Digital Quality Measures (dQMs)
- Terminology Resources (CodeSystems, ValueSets, etc.)

These assets serve as foundational specifications that guide behavior, structure, and expectations across domains like Reporting, Data Ingestion, and Evaluation.

Rather than owning executable services, this domain captures the processes, standards alignment, and content lifecycle required to ensure consistency and reusability across the system.

Requirements

Roles:

- Knowledge Artifact Administrator
- System Administrator
- Developer
- Quality Assurance Tester

Stakeholder Requirements

Requirement	Category
As a knowledge artifact administrator (specialization of systems admin), I can manage persistence, access, relationship and other characteristics of knowledge artifacts so that dependent functionality operates appropriately.	Process
As a knowledge artifact administrator, I can see what dependencies are missing for a loaded IG, so that I may find and load them and not have evaluation and/or validation fail at runtime.	Process
As a system administrator, I want to load FHIR Implementation Guides (IGs) that include Measures, Profiles, and Terminology artifacts, so that the platform can support standards-based evaluation and validation.	Process

As a quality assurance tester, I want to manage multiple versions of the same IG, to support testing changes to implementation guides without impacting other tenants and users.	Version Management
As a system administrator, I want to manage multiple versions of the same IG, to support incremental release of implementation guide updates across all tenants.	Version Management

System Requirements

Requirement	Category
As the system, I want to register FHIR IG artifacts in a shared repository, so that multiple dependent services can access them.	Process
As the system, I want to retrieve version-specific Measure artifacts from the knowledge artifact repository, so that reporting requirements can be confirmed.	Version Management
As the system, I want to access profile definitions from the loaded IGs, so that I can validate resource instances accurately.	Process
As the system, I want to load and reference ValueSet and CodeSystem definitions from IGs, so that I can confirm inform other services on the terminology needed to accurately evaluate and validate.	Process
As the system, I want to store IG version metadata and associate it with report evaluation records, so that report reconstruction uses the correct logic.	Report Reproducibility
As the system, I want to ensure IG updates do not overwrite previously loaded versions, so that all versions remain available for historical processing.	Version Management

Data Access

Type: Domain

Domain is responsible for storing and managing data (whether it be raw or normalized). It serves as a central repository for all data collected from various sources, providing a foundation for downstream data processing and analysis.

Key Concepts

- **Scalability:** Ability to handle large volumes of data.
- **Flexibility:** Support for diverse data types and formats.
- **Data Governance:** Policies and controls for managing data.

Responsibilities

- Ingesting data from various sources.
- Storing data in a scalable and cost-effective manner.
- Maintaining data integrity and availability.
- Providing access to raw data for downstream processing.

Requirements for Future Consideration

User Persona: Data Producer

- As a Data Producer, I want to write a new dataset to an Iceberg table so that it becomes available for downstream consumption with its associated metadata, including schema and provenance.
- As a Data Producer, I want to append new data to an existing Iceberg table so that the table reflects the latest information without disrupting ongoing queries.
- As a Data Producer, I want to overwrite existing data in an Iceberg table with a new version so that I can correct errors or update the dataset.
- As a Data Producer, I want to update specific records in an Iceberg table so that I can modify individual data points efficiently.
- As a Data Producer, I want to delete specific records from an Iceberg table so that I can remove outdated or incorrect information.
- As a Data Producer, I want to evolve the schema of an Iceberg table (e.g., add a new column) so that I can accommodate new data requirements without causing compatibility issues for existing data or consumers.
- As a Data Producer, I want to partition my data when writing to an Iceberg table based on specific columns so that queries can efficiently filter relevant data.
- As a Data Producer, I want to control the data file format (e.g., Parquet, ORC) used when writing to an Iceberg table so that I can optimize for performance and storage efficiency.
- As a Data Producer, I want to specify compression settings for the data files written to Iceberg so that I can balance storage costs and I/O performance.
- As a Data Producer, I want to track the lineage or provenance of the data I write to Iceberg so that consumers can understand its origin and transformations.
- As a Data Producer, I want to manage access control policies for the Iceberg tables I create so that only authorized users can write to them.
- As a Data Producer, I want to monitor the data writing process to Iceberg so that I can identify and resolve any issues.

- As a Data Producer, I want to rollback to a previous version of the table if a data writing operation introduces errors.

User Persona: Data Consumer (Analyst, Data Scientist, Application)

- As a Data Consumer, I want to query the current version of an Iceberg table so that I can access the latest and most accurate data for analysis or application use.
- As a Data Consumer, I want to query a specific historical version (snapshot) of an Iceberg table (time travel) so that I can analyze data as it existed at a particular point in time for auditing or reproducibility.
- As a Data Consumer, I want to filter data in an Iceberg table based on specific criteria so that I can focus on the relevant subset of information.
- As a Data Consumer, I want to join data from multiple Iceberg tables so that I can perform more complex analysis and gain deeper insights.
- As a Data Consumer, I want to understand the schema of an Iceberg table so that I can correctly interpret the data.
- As a Data Consumer, I want to discover available Iceberg tables and their descriptions so that I can find the data I need.
- As a Data Consumer, I want to understand the partitioning scheme of an Iceberg table so that I can optimize my queries.
- As a Data Consumer, I want to understand the data quality or validation rules associated with an Iceberg table so that I can assess the reliability of the data.
- As a Data Consumer, I want to be notified when an Iceberg table I'm interested in is updated so that I can react to new information.
- As a Data Consumer, I want to access the lineage or provenance information of an Iceberg table so that I can understand the data's journey.
- As a Data Consumer, I want to understand the access control policies on an Iceberg table so that I know what data I am authorized to access.

User Persona: Data Administrator/Engineer

- As a Data Administrator/Engineer, I want to create new Iceberg tables and define their initial schema and partitioning.
- As a Data Administrator/Engineer, I want to manage the metadata store (catalog) for Iceberg tables, ensuring its availability and integrity.
- As a Data Administrator/Engineer, I want to configure and manage data retention policies for Iceberg tables so that storage costs are optimized and compliance requirements are met.
- As a Data Administrator/Engineer, I want to perform compaction operations on Iceberg tables to optimize query performance and reduce the number of small files.
- As a Data Administrator/Engineer, I want to manage access control policies for Iceberg tables, granting and revoking permissions to different users and roles.
- As a Data Administrator/Engineer, I want to monitor the health and performance of Iceberg tables and the associated infrastructure.
- As a Data Administrator/Engineer, I want to perform data migration or upgrades of Iceberg tables as needed.
- As a Data Administrator/Engineer, I want to define and enforce data quality rules for Iceberg tables.
- As a Data Administrator/Engineer, I want to configure and manage data skipping strategies for Iceberg tables to improve query efficiency.
- As a Data Administrator/Engineer, I want to integrate Iceberg with various compute engines and data processing frameworks.
- As a Data Administrator/Engineer, I want to implement backup and recovery strategies for Iceberg metadata and data.

Compliance

Type: Domain

Responsible for ensuring that Link operates within defined policies. This includes both operational compliance (e.g., security controls auditing) and data compliance (e.g., validating data syntax and semantics against prescribed profiles and standards).

Key Concepts

- Data Governance: Setting policy and assuring compliance

Requirements

Stakeholder Requirements

- Clinical data shall meet USCore standard compliance requirements
- FHIR implementation guides containing profiles shall be loadable for validation purposes
- Clinical data shall be validated prior to submission
- Open source validation technologies shall be used for FHIR standard compliance validation
- Validation results shall be persisted for future reference
- Validation results shall be categorized to facilitate stakeholder review
- Validation results and categories shall be visualized with drill-down capability
- Uncategorized validation results shall be identifiable for category creation

System Requirements

- The system shall validate all clinical data against USCore profiles
- The system shall support importing FHIR implementation guides containing validation profiles
- The system shall execute validation checks on clinical data before allowing submission
- The system shall use HAPI FHIR validator for FHIR standard compliance validation
- The system shall persist validation results in a queryable database
- The system shall support creation and management of validation result categories
- The system shall provide visualization of validation results with category-to-detail navigation
- The system shall maintain an "uncategorized" category for validation results without defined categories

Services

Admin UI (front-end)

Type: Service

Overview

The Admin UI is an Angular-based front-end application that provides a comprehensive interface for managing the Link platform, configuring tenants, and monitoring report generation and submission workflows.

System-wide Functionality

- **User Management**: Allows administrators to manage user accounts, including creation, editing, and deletion. Currently, the system supports a single "admin" role for platform-wide access.
- **Measure Management**: Upload and manage FHIR-based measure definitions. Administrators can upload measure bundles in JSON format and explore related artifacts.
- **Validation Profile (IG) Management**: Supports the management of FHIR Implementation Guides (IGs). Administrators can upload IGs as `.tgz` packages and view their terminology dependencies and package details.
- **Terminology Management**: Allows for viewing and filtering terminology resources, such as CodeSystems and ValueSets, that are utilized by the platform's terminology service.
- **Integration Testing Support**: Offers tools to initiate and track integration tests, enabling administrators to validate service connectivity and workflow correctness within the environment.

Tenant Configuration

- **Basic Configuration**: Administrators can define tenant-specific settings, including Facility IDs, names, and timezones.
- **Census Scheduling and Acquisition Methods**: Supports configuring how patient census data is acquired, including defining schedules and enabling/disabling automated census retrieval.
- **Measure Association & Scheduling**: Enables associating specific measures with a tenant and configuring automated reporting schedules (Daily, Weekly, or Monthly cadences).
- **Data Sources & Acquisition**
  :
  - **FHIR Query Config**: Configuration of FHIR-based data acquisition parameters.
  - **FHIR List Config**: Management of patient list-based acquisition settings.
  - **Query Plans**: Definition and management of query plans used to retrieve clinical data from data sources.

- **Normalization Operations**: Configuration of transformation rules and operations applied to data during the normalization phase.

Tenant & Report Viewing

- **Tenant Dashboard**: Provides a high-level view of a tenant's configuration and a list of all reports generated for that tenant.
- **Report Details**
  :
  - **Report Summary**: Visual representation of report status using donut charts for Measure IP counts, Reporting Status, and Submission Status.

    - **Patient List**: A detailed table of all patients included in a report, showing individual reporting and submission statuses.
    - **Pre-qualification (Validation)**: Detailed access to validation results, categorized by issue type (e.g., missing profiles or terminology warnings), allowing for quick identification of data quality issues.

- **Report Downloading**: Enables download of generated report packages for manual delivery, archival, or verification once a report has been submitted.

Data Acquisition Log Analysis

- **Log Access**: Integration with data acquisition logs allows administrators to analyze acquisition statistics and troubleshoot data retrieval issues directly from the report view.
- **Analysis Tools**: Provides insights into the success and failure rates of data acquisition steps for a given report.

Generating Adhoc Reports

- **On-demand Generation**: Allows administrators to manually trigger report generation for a specific facility, reporting period, and set of measures.
- **Patient Scoping**: Supports restricting adhoc reports to a specific cohort by manually entering patient IDs or uploading a CSV file of patient identifiers.
- **Submission Control**: Includes an option to bypass automatic submission, enabling administrators to generate and review reports before they are sent to external entities.

Volumes

Volume	Mount Path	Sub-path
Azure Storage Account	/usr/share/nginx/html/assets/app.config.local.json	app.config.local.json

Configuration

An app.config.local.json file is used to configure the Admin UI service at deployment time, overriding the defaults in app.config.json.

```
{
  "baseApiUrl": "<ADMIN-BFF-BASE-URL>/api",
  "authRequired": true,
  "oauth2": {
    "enabled": true,
    "issuer": "...",
    "clientId": "...",
    "scope": "openid profile email",
    "responseType": "code"
  }
}
```

This configuration file can be customized at deployment time and mounted in the /assets folder, or it can be overridden by ENV variables in a docker deployment.

Authentication

The Admin UI supports authentication using:

- OAuth2 via an identity provider (e.g., Azure AD, Keycloak), enabling secure login with centralized identity and access management.
- Token Generation via AdminBFF /api/login, which provides session tokens for downstream API access when deployed in a simplified or standalone context.

Environment Variables

ENV variable	JSON Property	Description	Default Value	Required
LINK_BASE_API_URL	baseApiUrl	The base URL for the Admin BFF API.	http://localhost	Yes
LINK_AUTH_REQUIRED	authRequired	Indicates whether authentication is required for the Admin UI.	true	Yes
LINK_OAUTH2_ENABLED	oauth2.enabled	Indicates whether OAuth2 authentication is enabled.	false	Yes
LINK_OAUTH2_ISSUER	oauth2.issuer	The issuer URL for the OAuth2 provider.	null	If oauth2 enabled
LINK_OAUTH2_CLIENT_ID	oauth2.clientId	The client ID for the OAuth2 application.	null	If oauth2 enabled
LINK_OAUTH2_SCOPE	oauth2.scope	The scope for the OAuth2 application.	openid profile email	If oauth2 enabled
LINK_OAUTH2_RESPONSE_TYPE	oauth2.responseType	The response type for the OAuth2 application.	code	If oauth2 enabled
GRAFANA_URL	grafanaUrl	URL for the Grafana UI.	http://localhost:3000	Yes

KAFKA_URL	kafkaUrl	URL for the Kafka UI.	http://localhost:9095	If Kafka UI is available

Local Development with Proxy Configuration

To facilitate local development and testing, the Admin UI can be configured to proxy API requests to a local or remote instance of the Admin BFF service. This is achieved by setting up a proxy configuration file.

```
{
  "/api": {
    "target": "http://localhost:5218",
    "secure": false,
    "changeOrigin": true,
    "logLevel": "debug"
  }
}
```

To use the proxy configuration, start the Angular development server with the following command:

```
ng serve --proxy-config proxy.conf.json --configuration=development
```

Admin Backend-for-Frontend

Type: Service

Overview

The AdminBFF (Backend-for-Frontend) service is the administrative gateway that bridges the Angular-based front-end application with the internal service ecosystem. It serves as the single public-facing backend entry point for all administrative operations, providing a tightly controlled interface that enforces security, authorization, and request routing policies.

- **API Gateway for Front-End**: Acts as the dedicated backend interface for the admin UI, simplifying and consolidating access to internal services.
- **Reverse Proxy via YARP**: Uses YARP (Yet Another Reverse Proxy) to intelligently route and proxy requests to internal backend services. This enables seamless service composition, service path rewriting, and centralized traffic control.
- **Authorization Enforcement**: Validates that all incoming requests are made by authenticated users with the proper roles and scopes. It serves as the first and last line of defense before internal services are reached, ensuring that only authorized actions are executed.
- **Facade for Backend Complexity**: Exposes a front-end-optimized API layer, reducing the need for the UI to manage low-level backend details or coordinate multiple service calls directly.
- **Config File Serving for Admin UI**: AdminBFF also serves a runtime configuration file used by the Angular-based Admin UI service. This allows environment-specific values (e.g. API base URLs, feature flags) to be injected at deployment time.

The "Config File Serving for Admin UI" is only applicable when AdminBFF and AdminUI are deployed together as a single instance, such as in containerized deployments or bundled cloud environments.

Configuration

Property	Type	Description
Redis:Enabled	boolean	If enabled, Redis will be used to cache account information associated with authenticated users.

Tenant Service

Type: Service

Overview

The Tenant service is the entry point for configuring a tenant into Link Cloud. The service is responsible for maintaining and generating events for the scheduled measure reporting periods that the tenant is configured for. These events contain the initial information needed for Link Cloud to query resources and perform measure evaluations based on a specific reporting period.

The Tenant service uses Quartz with ADO Job Stores to manage jobs across multiple instances of the service, primarily for producing ReportScheduled Kafka events.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- SQL Server Database

Custom Configuration

Name	Value	Secret?
MeasureConfig__CheckIfMeasureExists	true or false	No

Soft Delete

The Tenant service supports soft deletion of facilities. Rather than permanently removing a facility record, the soft delete endpoint marks the facility with an `isDeleted` flag set to `true`. This preserves the facility's

configuration and historical data while effectively removing it from active use.

DELETE /api/Facility/softDelete/{facilityId}

Soft deletes a facility by setting its `isDeleted` flag to `true`.

Parameters:

- `facilityId` (path, required): The unique identifier of the facility to soft delete.

Response:

- `200 OK`: Facility soft deleted successfully. Returns the updated facility configuration.
- `400 Bad Request`: If the request is invalid.
- `404 Not Found`: If no facility exists for the specified facilityId.
- `500 Internal Server Error`: If an error occurs during processing.

Features and Functionality

Key Management

Keys are managed independently by the tenant service, and then associated with tenants. These keys are used for signing authentication JWTs that are passed to /token endpoints at the EHRs. EHRs check that the JWTs are signed using a key found in the JWKS endpoint exposed by Link.

- Key Registry: A global repository where all JWT signing keys are stored and managed. Each key includes metadata such as kid, algorithm, creation timestamp, status (active, disabled, expired), and an optional primary flag.
- Lifecycle Operations:
  - Create: Keys can be generated through the admin API/UI with algorithm selection and optional designation as primary.
  - Disable/Delete: Keys no longer in use can be deactivated or removed without disrupting tenant authentication, unless actively in use. This can occur when needing to rotate a key.
  - Rotating Keys:
    1. Create a new key
    2. Associate the new key with tenants
    3. Delete the old key.

- Tenant Association (handled separately):
  - Tenants reference keys by kid through the Tenant Service configuration interface.
  - A tenant may use a shared or unique key from the registry.
  - Changing a tenant's associated key does not impact the underlying key object—it only changes the reference.

- JWKS documents are managed automatically
  - When keys change the JWKS document is re-generated
  - The JWKS document is stored on the file system

- The JWKS document on the file system is exposed via proxy to external parties for verification (such as EHRs)

The following sequence diagram illustrates how the keys are used during the authentication requests to EHRs:

```
sequenceDiagram
    participant L as Link
    participant J as JWKS Endpoint
    participant E as EHR Vendor

    L->>L: Create JWT
    L->>L: Sign JWT (with kid)
    L->>J: Publish key to JWKS
    L->>E: POST /token (with JWT)
    E->>J: GET /.well-known/jwks.json
    J-->>E: Return JWKS document
    E->>E: Verify signature using kid
    E-->>L: Return access token
```

Census Service

Type: Service

Overview

The Census service is primarily responsible for maintaining a tenants admit and discharge patient information needed to determine when a patient is ready for reporting. To accomplish this, the Census service has functionality in place to request an updated FHIR List of recently admitted patients. The frequency that the request is made is based on a Tenant configuration made in the Census service.

The system routinely gathers census data by leveraging multiple methods compatible with hospital EHRs. This document provides an overview of each acquisition method, the data elements tracked, and future considerations for expanding data persistence.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Event Sourcing Workflow

The Census service consumes `PatientListsAcquired` events from Kafka. When a new event is received, the `PatientListsAcquiredListener` processes the payload, updating census records for the specified facility. This enables downstream reporting and data acquisition.

Workflow Steps:

1. **Scheduling:** A scheduled job (per tenant configuration) triggers a request for updated patient lists.
2. **Data Acquisition:** The system sends a `DataAcquisitionRequested` event, prompting the acquisition of FHIR patient lists.
3. **Event Production:** Upon successful acquisition, a `PatientListsAcquired` event is produced, containing the list of patient FHIR IDs.
4. **Census Update:** The Census service consumes the event and updates its records accordingly.

Event Payload Structures

PatientCensusScheduled Event

```
{
  "facilityId": "string",
  "scheduledTime": "string"
}
```

- **facilityId**: Facility identifier.
- **scheduledTime**: Scheduled time for census acquisition.

FHIRListAdmitPayload Event

```
{
  "patientId": "string",
  "admitDate": "string (ISO 8601 datetime)"
}
```

- **patientId**: The unique FHIR ID of the patient being admitted.
- **admitDate**: The datetime when the patient was admitted.
- **payloadType**: Automatically set to "FHIRListAdmit" (not included in JSON).

This event is used to track patient admissions detected through FHIR list processing. When processed, it creates a new PatientEncounter record in the system with the admission date and patient identifier.

FHIRListDischargePayload Event

```
{
  "patientId": "string",
  "dischargeDate": "string (ISO 8601 datetime)"
}
```

- **patientId**: The unique FHIR ID of the patient being discharged.
- **dischargeDate**: The datetime when the patient was discharged.
- **payloadType**: Automatically set to "FHIRListDischarge" (not included in JSON).

This event is used to track patient discharges detected through FHIR list processing. When processed, it updates an existing PatientEncounter record with the discharge date, indicating the patient's encounter has concluded.

Both FHIR list payloads are critical components of the patient census tracking system, enabling the service to maintain accurate patient encounter records as patients enter and exit the healthcare facility. These events, along with other event types, help build a complete picture of patient movement for reporting purposes.

PatientListsAcquired Event

```
{
  "reportTrackingId": "string",
  "patientLists": [
    {
      "listType": "string" // Acceptable values: "Admit", "Discharge",
      "timeFrame": "string" // Acceptable values: "LessThan24Hours", "Between24To48Hours", "MoreThan48Hours",
      "patientIds": ["string"]
    }
  ]
}
```

- **facilityId**: The unique identifier for the facility.

- **patientIds**: Array of FHIR patient IDs.
- **acquisitionTimestamp**: ISO 8601 timestamp of acquisition.

PatientCensusScheduled Event

```
{
  "facilityId": "string",
  "scheduledTime": "string"
}
```

- **facilityId**: Facility identifier.
- **scheduledTime**: Scheduled time for census acquisition.

API Documentation

Census Configuration Controller

The Census service exposes REST endpoints to manage tenant-specific census configuration.

GET /census/config/{facilityId}

Retrieves the census configuration for a specific facility.

Parameters:

- `facilityId` (path, required): The unique identifier of the facility.

Response:

- `200 OK`: Returns the census configuration.
- `404 Not Found`: If no configuration exists for the specified facilityId.

Response Model:

```
{
  "facilityId": "string",
  "scheduledTrigger": "string",
  "enabled": boolean
}
```

POST /census/config

Creates a new census configuration.

Request Body:

```
{
  "facilityId": "string",
  "scheduledTrigger": "string",
  "enabled": boolean
}
```

- facilityId (required): The unique identifier of the facility.
- scheduledTrigger (required): A CRON expression defining the schedule for census acquisition.
- enabled (optional, default: true): Whether the census acquisition is enabled.

Response:

- 201 Created: Configuration created successfully.
- 400 Bad Request: If the request is invalid.

Response Model:

```
{
  "id": "string"
}
```

PUT /census/config/{facilityId}

Updates an existing census configuration.

Parameters:

- facilityId (path, required): The unique identifier of the facility.

Request Body:

```
{
  "facilityId": "string",
  "scheduledTrigger": "string",
  "enabled": boolean
}
```

Response:

- 200 OK: Configuration updated successfully.
- 404 Not Found: If no configuration exists for the specified facilityId.

DELETE /census/config/{facilityId}

Deletes a census configuration.

Parameters:

- facilityId (path, required): The unique identifier of the facility.

Response:

- 200 OK: Configuration deleted successfully.
- 404 Not Found: If no configuration exists for the specified facilityId.

Census Controller

Endpoints for retrieving patient census data and patient lists.

GET /api/census/{facilityId}/history/admitted

Gets the admitted patients for a facility within a date range. Returns a FHIR List resource containing patient references.

Parameters:

- facilityId (path, required): The unique identifier of the facility.
- startDate (query, required): Start date for the date range filter.
- endDate (query, required): End date for the date range filter.

Response:

- 200 OK: Returns a FHIR List resource with admitted patients.
- 404 Not Found: If no patients found for the specified facility.
- 400 Bad Request: If invalid arguments are provided.
- 500 Internal Server Error: If an error occurs during processing.

Response Model: Returns a FHIR List resource with the following structure:

- status: Current
- mode: Snapshot
- extension: Contains applicable period with start and end dates
- entry[]: Array of patient references in format "Patient/{patientId}"

Patient Encounters Controller

Endpoints for managing and retrieving patient encounter materialized views.

GET /api/census/patient-encounters/current

Returns the current materialized view state for patient encounters for a given facility.

Parameters:

- facilityId (query, required): The unique identifier for the facility.
- correlationId (query, optional): Optional correlation ID to filter patient encounters.
- sortBy (query, optional): Field to sort by (e.g., "CorrelationId", "AdmitDate").
- sortOrder (query, optional): Sort order (Ascending or Descending).
- pageSize (query, optional): Number of records per page (default: 10).
- pageNumber (query, optional): Page number to retrieve (default: 1).

Response:

- 200 OK: Returns a paged list of current patient encounters.
- 400 Bad Request: If facilityId is not provided.
- 500 Internal Server Error: If an error occurs during processing.

Response Model:


```
{
  "records": [
    {
      "id": "string",
      "correlationId": "string",
      "facilityId": "string",
      "medicalRecordNumber": "string",
      "admitDate": "datetime",
      "dischargeDate": "datetime",
      "encounterType": "string",
      "encounterStatus": "string",
      "encounterClass": "string",
      "createDate": "datetime",
      "modifyDate": "datetime",
      "patientVisitIdentifiers": [],
      "patientIdentifiers": []
    }
  ],
  "pageNumber": 1,
  "pageSize": 10,
  "totalCount": 100
}
```


GET /api/census/patient-encounters/historical

Returns an ad hoc generated materialized view state for patient encounters for a given facility as of a specific date.

Parameters:

- facilityId (query, required): The unique identifier for the facility.
- correlationId (query, optional): Optional correlation ID to filter patient encounters.
- dateThreshold (query, required): The date as of which to generate the historical view.
- sortBy (query, optional): Field to sort by (e.g., "CorrelationId", "AdmitDate").
- sortOrder (query, optional): Sort order (Ascending or Descending).

- pageSize (query, optional): Number of records per page (default: 10).
- pageNumber (query, optional): Page number to retrieve (default: 1).

Response:

- 200 OK: Returns a paged list of patient encounters as of the specified date.
- 400 Bad Request: If facilityId or dateThreshold is not provided.
- 500 Internal Server Error: If an error occurs during processing.

Response Model: Same as current patient encounters endpoint.

POST /api/census/patient-encounters/rebuild

Deletes and rebuilds the materialized view records for a given facility.

Parameters:

- facilityId (query, required): The unique identifier for the facility.
- correlationId (query, optional): Optional correlation ID to filter which materialized view to rebuild.

Response:

- 202 Accepted: If the rebuild is successful.
- 400 Bad Request: If facilityId is not provided.
- 500 Internal Server Error: If an error occurs during processing.

Patient Events Controller

Endpoints for managing and retrieving patient events.

GET /api/census/patient-events

Returns all events stored in the patient events data store for the given facility.

Parameters:

- facilityId (query, required): The unique identifier for the facility.
- correlationId (query, optional): Optional correlation ID to filter events.
- startDate (query, optional): Optional start date to filter events.
- endDate (query, optional): Optional end date to filter events.
- sortBy (query, optional): Field to sort by.
- sortOrder (query, optional): Sort order (Ascending or Descending).
- pageSize (query, optional): Number of records per page (default: 10).
- pageNumber (query, optional): Page number to retrieve (default: 1).

Response:

- 200 OK: Returns a paged list of patient events.
- 404 Not Found: If no patient events found for the facility.
- 400 Bad Request: If facilityId is not provided.
- 500 Internal Server Error: If an error occurs during processing.

Response Model:

```
{
 "records": [
  {
   "id": "string",
   "facilityId": "string",
   "correlationId": "string",
   "sourcePatientId": "string",
   "sourceVisitId": "string",
   "medicalRecordNumber": "string",
   "eventType": "string",
   "payload": {},
   "sourceType": "string"
  }
 ],
 "pageNumber": 1,
 "pageSize": 10,
 "totalCount": 100
}
```

DELETE /api/census/patient-events/{id}

Soft deletes a patient event and rebuilds the materialized view for the related correlation id.

Parameters:

- id (path, required): The unique identifier of the patient event to delete.

Response:

- 202 Accepted: If deletion is successful.
- 400 Bad Request: If patient event ID is not provided.
- 500 Internal Server Error: If an error occurs during processing.

DELETE /api/census/patient-events/visit/{correlationId}

Soft deletes the patient event store for the given correlation id and removes the corresponding materialized view.

Parameters:

- correlationId (path, required): The correlation ID for the visit to delete.

Response:

- 202 Accepted: If deletion is successful.
- 400 Bad Request: If correlation ID is not provided.
- 500 Internal Server Error: If an error occurs during processing.

Technical Implementation

The Census service is implemented as a .NET 8.0 web application with the following key components:

Core Technologies

- **Framework**: .NET 8.0
- **Database**: Microsoft SQL Server via Entity Framework Core 8.0.3
- **Messaging**: Kafka (Confluent.Kafka 2.*)
- **Scheduling**: Quartz.NET 3.*
- **API Documentation**: Swagger/OpenAPI
- **Logging**: Serilog with Grafana Loki integration

Database Schema

The Census service uses Entity Framework Core with SQL Server for data persistence. The database schema includes:

1. **CensusConfiguration**: Stores facility-specific census configurations

    - Primary Key: FacilityId (string)
    - ScheduledTrigger (string): CRON expression for scheduling
    - Enabled (bool): Flag to enable/disable census acquisition

2. **PatientCensus**: Tracks patient census information

    - Primary Key: Id (Guid)
    - FacilityId (string): Facility identifier
    - PatientId (string): FHIR patient identifier
    - AdmissionDate (DateTime?): Optional admission date
    - DischargeDate (DateTime?): Optional discharge date
    - LastUpdated (DateTime): Timestamp of last update

Scheduled Jobs

The Census service uses Quartz.NET with ADO Job Stores for scheduling the census acquisition jobs based on tenant configurations. Each job:

1. Reads the facility-specific configuration
2. Triggers the census acquisition process
3. Emits the appropriate events
4. Updates the database with the results

Event Processing

The Census service processes incoming Kafka events via dedicated listeners and produces events when census data is acquired or scheduled.

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Kafka Consumer Retry Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- SQL Server Database Configuration

Application Settings

The Census service uses the following configuration settings:

```
{
  "ServiceInformation": {
    "Name": "Census",
    "Version": "0.5.0"
  },
  "ConnectionStrings": {
    "CensusDatabase": "[connection-string]"
  },
  "Kafka": {
    "bootstrap.servers": "localhost:9092",
    "group.id": "census-service",
    "auto.offset.reset": "earliest"
  },
  "KafkaTopics": {
    "PatientListsAcquired": "patient-lists-acquired",
    "PatientCensusScheduled": "patient-census-scheduled",
    "AuditableEventOccurred": "audit-events",
    "PatientEvent": "patient-events"
  },
  "Quartz": {
    "quartz.scheduler.instanceName": "Census-Scheduler"
  },
  "Logging": {
    "LogLevel": {
```

```
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Serilog": {
    "Using": [
      "Serilog.Sinks.Console",
      "Serilog.Sinks.Grafana.Loki"
    ],
    "MinimumLevel": {
      "Default": "Information",
      "Override": {
        "Microsoft": "Warning",
        "System": "Warning"
      }
    },
    "WriteTo": [
      {
        "Name": "Console"
      },
      {
        "Name": "GrafanaLoki",
        "Args": {
          "uri": "[loki-url]",
          "labels": [
            {
              "key": "app",
              "value": "census-service"
            }
          ]
        }
      }
    ],
    "Enrich": [
      "FromLogContext",
      "WithMachineName",
      "WithThreadId",
      "WithSpan"
    ]
  }
}
```

Features and Functionality

Link includes a **Census Management** module designed to acquire, evaluate, and maintain a real-time census of patients actively or recently treated within a hospital system. This module supports the submission of required patient data to governing bodies (such as NHSN) per established reporting criteria.

The system routinely gathers census data by leveraging multiple methods compatible with hospital EHRs. This document provides an overview of each acquisition method, the data elements tracked, and future considerations for expanding data persistence.

Census Data Acquisition Methods

1. FHIR Standard - List Resource

The **FHIR List Resource** method is one of the primary approaches for acquiring patient lists from hospital EHR systems. In systems like Epic, patient lists are generated through proprietary queries within the EHR, associated with the FHIR List resource for access through FHIR integrations.

- **Endpoint**: GET /List/<listId>
- **Functionality**: Queries the EHR for patient lists identified by a listId.
- **Tenant Configurability**: The listId is configurable per tenant within Link, allowing each institution to define the patient population that constitutes their census.
- **Applicability**: This method supports census management for any EHR that utilizes FHIR Lists representing relevant patient groups.
- **Rules**
  : The following rules are enforced by Census when processing FHIR List resources:
  - Admission lists are processed before discharge lists.
  - If a patient appears on a discharge lists with no corresponding admit event, an admission event is created and then the patient is discharged.
  - If a patient is admitted and then disappears off of all incoming lists, the patient is discharged.

2. FHIR Standard - Bulk FHIR Implementation Guide

The **Bulk FHIR** method allows Link to acquire patient data via batch processing, useful for large patient groups in systems that support flexible querying.

- **Endpoint**: Bulk FHIR $export request with groupId
- **Process**
  :
  - Link initiates a $export request for patient data by groupId.
  - The export process is monitored and polled routinely until completion.
  - Upon completion, patient resources are retrieved, and the FHIR ID of each patient is extracted and stored.

- **Tenant Configurability**: Each tenant configures a unique groupId corresponding to their desired patient group.

3. ADT Feeds (Under Exploration)

Link is evaluating the feasibility of using **ADT feeds**—specifically for admission, discharge, and cancellation events—to dynamically manage census data. This approach would offer real-time census updates and potentially enhance the accuracy and responsiveness of the census.

Scheduling and Frequency

Both the FHIR List and Bulk FHIR methods utilize a configurable scheduling system, allowing each tenant to define query intervals. The scheduling system is based on CRON patterns, ensuring Link can automatically query the EHR at specified times to maintain an updated census.

Census uses Quartz with ADO Job Stores to manage jobs across multiple instances of the service, primarily for scheduling Patient List retrieval and Retry events.

Data Persistence and Tracking

Currently, Link persists only the **FHIR ID** of each patient. This identifier allows Link to accurately track patients across census updates without storing additional demographic information.

Future Considerations

In the interest of enhancing the user interface, Link is considering storing additional data elements, such as the **patient name** associated with each FHIR ID. This would provide users with meaningful patient identifiers, facilitating easier navigation and record management within the Link UI.

Build and Deployment

The Census service is containerized using Docker with Linux as the target OS. It is designed to be deployed as part of a larger microservices architecture, with dependencies on other Link services for complete functionality.

Submission Service

Type: Service

Overview

The Submission service is responsible for submitting a tenant's generated reporting content to a configured destination. Currently, the service submits reports to a configured Azure Blob Storage destination. For local testing, the Azurite Docker image (Link Here) can be used as a local instance of Azure Blob Storage.

File	Description	Multiple Files?
Aggregate	A MeasureReport resource that contains references to each patient evaluation for a specific measure	Yes, one per measure
Manifest	A Bundle resource that contains references to each of the submission files (Patient, Aggregate, etc). This file is extended for System State Manifest support.	No
Patient List	A List resource of all patients that were admitted into the facility during the reporting period	No
Device	A Device resource that details the version of Link Cloud that was used	No
Organization	An Organization resource for the submitting facility	No
Other Resources	A Bundle resource that contains all of the shared resources (Location, Medication, etc) that are referenced in the patient Measure Reports	No
Patient	A Bundle resource that contains the MeasureReports and related resources for a patient	Yes, one per evaluated patient

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Kafka Consumer Retry Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- Mongo Database

Custom Configurations

Name	Value	Description	Secret?
SubmissionServiceConfig__SubmissionDirectory	<string>	The location of where to store submission files until they are ready to be submitted. i.e. /data/Submission	No
SubmissionServiceConfig__PatientBundleBatchSize	1	The number of patients to process during submission in parallel (as seperate threads)	No
SubmissionServiceConfig__MeasureNames__0__Url	<string>	URL of measure	No
SubmissionServiceConfig__MeasureNames__0__MeasureId	<string>	ID of measure	No
SubmissionServiceConfig__MeasureNames__0__ShortName	<string>	Short name of the measure (used in building submission file name)	No
Features__DownloadReportEnabled	<boolean>	Whether or not the download report feature is enabled. Defaults to false.	No

Functionality

The Link Submission service is responsible for securely transmitting compiled measure reports and their associated clinical data to an external Blob Storage container. The payload consists of Newline-Delimited JSON (.ndjson) files representing FHIR resources.

Structure and Timing

The transmission process is triggered asynchronously by the SubmitPayload Kafka topic. When a payload is ready for submission:

1. The Submission service's `SubmitPayloadListener` picks up the message. The `PayloadType` on the value of the event will determine whether a patient report or a facility summary manifest file is being sent to the external endpoint. Patient report submissions are immediately submitted when a patient has been evaluated and validated for all measures within a reporting period.
2. The service streams the pre-compiled `.ndjson` content. It optimizes this by attempting to read directly from the internal blob storage (where the Report service initially placed it). If the internal client is unavailable or the file cannot be found, it falls back to retrieving the bundle via the Report service's REST API.
3. The content is streamed directly to the external Blob Storage container.
4. The service emits a `PayloadSubmitted` Kafka event upon successful transfer, and logs an Audit Event for traceability.

This mechanism allows the system to scale and asynchronously write large files without holding up upstream measure evaluation processes or creating excessive memory pressure.

File Hierarchy and Naming Conventions

Data submitted to the external blob storage is organized hierarchically. A "folder" (blob prefix) is created per report schedule, and the related `.ndjson` files are placed within it.

Folder Naming Convention

The folder path is constructed dynamically and follows this naming pattern: `\{BlobRoot}/\{facilityId}_\{dqms}_\{reportingStartDate}_\{scheduleId}/`

- **facilityId**: The submitting organization's Facility ID (lowercase).
- **dqms**: A + separated list of shortened, sorted measure names (e.g., ach+hypo).
- **reportingStartDate**: The start date of the reporting period formatted as yyyyMMdd.
- **scheduleId**: A sanitized, URL-safe version of the Report Schedule GUID.

Example Folder: `98234_ach+hypo_20240101_e6a8d3..._/`

File Naming Conventions

Inside the submission folder, the structured files are named as follows:

- **Aggregate Data**: manifest.ndjson
- **Patient Data**: patient-{patientId}.ndjson (One file is generated per patient)

Aggregate Level Information (manifest.ndjson)

The manifest.ndjson file contains the summary-level information for the report bundle. It uses the application/x-ndjson content type, where each line is a fully valid JSON object representing a FHIR

resource.

The structure inside the aggregate manifest includes:

1. **MeasureReport**: Represents the summary-level (population) aggregation of all individual patient measure reports. It contains population counts and scoring.
2. **List**: A FHIR List resource that references each individual patient who met the specific dQM criteria (e.g., Initial Population, Numerator, Denominator).
3. **Device**: A resource representing the "Submitting Device", which includes version information and software details about the Link platform performing the submission.
4. **Organization**: A resource representing the submitting organization.
5. **OperationOutcome**: A resource representing the patient id's that were submitted to the external endpoint, but failed validation. A single resource would be generated with multiple issues representing each failed patient.

Individual Patient Information (patient-{patientId}.ndjson)

Each patient whose data is being submitted has their own dedicated .ndjson file. Like the manifest, each line is a serialized, valid FHIR resource.

The structure of the individual patient file logically flows as:

1. **MeasureReport**: Starts with the individual-level MeasureReport. This details exactly which populations the patient fell into and points to the specific pieces of clinical data that served as evidence.
2. **Patient**: The primary FHIR Patient resource describing the individual's demographics.
3. **Clinical Data**: The remainder of the .ndjson lines consist of the evaluated clinical resources, such as Observation, Condition, Encounter, MedicationRequest, etc., that were pulled from the EHR and factored into the measure calculation.
4. **OperationOutcome**: This resource would show in the patient file if they fail validation.

Submission Storage Hierarchy

Below is an example of the hierarchical structure of a facility's Azure Blob Storage submission for a facility NSHN ACH Monthly report:

```
98234_achm_20260201_a1322b82-37b4-4cb2-91b0-bcad0a703086
  manifest.ndjson
    MeasureReport
    List
    Device
    OperationOutcome
  patient-jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN.ndjson
    MeasureReport
    Patient
    Encounter
    Location
    Encounter
    Location
    Medication
  patient-PfN97mRomyKGqn9vgGDXpp5u97fiE9v6iXxeyNdXNtn6J.ndjson
    MeasureReport
    Patient
```

- Encounter
- Location
- Observation
- DiagnoticReport
- OperationOutcome

Manifest.ndjson Example

Below is an example of the FHIR resources that are added to the manifest.ndjson file:

{"resourceType":"Organization","id":"af0e74a6-24ca-4d22-abe3-12c3bf96744c","meta":{"profile":["https://www.cdc.gov/nhsn/nhsn-measures/StructureDefinition/nhsn-submitting-organization"]},"identifier":[{"system":"https://www.cdc.gov/nhsn/OrgID","value":"MyFacility-1775588582125"}],"active":true,"type":[{"coding":[{"system":"http://terminology.hl7.org/CodeSystem/organization-type","code":"prov","display":"Healthcare Provider"}]}],"name":"MyFacility-1775588582125","telecom":[{"extension":[{"url":"http://hl7.org/fhir/StructureDefinition/data-absent-reason","valueCode":"unknown"}]}],"address":[{"extension":[{"url":"http://hl7.org/fhir/StructureDefinition/data-absent-reason","valueCode":"unknown"}]}]}
{"resourceType":"Device","id":"d5be3f1e-c031-4ebb-b202-92ba87e3f1ee","deviceName":[{"name":"NHSNLink"}],"version":[{"value":""}]}
{"resourceType":"List","id":"aaa5024e-de6a-4f27-8a35-da977ca404e0","extension":[{"url":"http://www.cdc.gov/nhsn/fhirportal/dqm/ig/StructureDefinition/link-patient-list-applicable-period-extension","valuePeriod":{"start":"2025-11-01T05:00:00+00:00","end":"2025-12-01T05:59:59+00:00"}}],"status":"current","mode":"snapshot","entry":[{"item":{"reference":"Patient/PfN97mRomyKGqn9vgGDXpp5u97fiE9v6iXxeyNdXNtn6J"}},{"item":{"reference":"Patient/jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN"}}]}
{"resourceType":"MeasureReport","id":"0f751f54-f405-4060-94d5-a6cb7038b3e5","meta":{"profile":["http://www.cdc.gov/nhsn/fhirportal/dqm/ig/StructureDefinition/subjectlist-measurereport"]},"contained":[{"resourceType":"List","id":"initial-population-list","status":"current","mode":"snapshot","entry":[{"item":{"reference":"MeasureReport/a775db5f-f00a-4a05-aa1e-b979cad13a19"}},{"item":{"reference":"MeasureReport/f2ec24e5-2342-4eb1-9c94-1c2096d37d87"}}]}],"status":"complete","type":"subject-list","measure":"http://www.cdc.gov/nhsn/fhirportal/dqm/ig/Measure/NHSNAcuteCareHospitalMonthlyInitialPopulation|1.0.0-dev","date":"2026-04-24T14:17:18.9758292+00:00","reporter":{"reference":"Organization/af0e74a6-24ca-4d22-abe3-12c3bf96744c"},"period":{"start":"2025-11-01T05:00:00+00:00","end":"2025-12-01T05:59:59+00:00"},"group":[{"population":[{"code":{"coding":[{"system":"http://terminology.hl7.org/CodeSystem/measure-population","code":"initial-population","display":"Initial Population"}]},"count":221,"subjectResults":{"reference":"#initial-population-list"}}]}]}
{"resourceType":"OperationOutcome","id":"e18b348c-2cd9-4431-b3b9-5fba3550dc0b","issue":[{"severity":"error","code":"invalid","diagnostics":"Validation failed for patient jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN"},{"severity":"error","code":"invalid","diagnostics":"Validation failed for patient PfN97mRomyKGqn9vgGDXpp5u97fiE9v6iXxeyNdXNtn6J"}]}

Patient.ndjson Example

Below is an example of a submitted patient.ndjson file for a patient in the manifest.ndjson file shown above:

{"resourceType":"MeasureReport","id":"a775db5f-f00a-4a05-aa1e-b979cad13a19","extension":[{"url":"http://hl7.org/fhir/5.0/StructureDefinition/extension-MeasureReport.population.description","valueString":"The Acute Care Hospital Monthly Initial Population includes all encounters for patients of any age in an ED, observation, or inpatient location or all encounters for patients of any age with an ED, observation, inpatient, or short stay status during the measurement period."},{"url":"http://hl7.org/fhir/5.0/StructureDefinition/extension-MeasureReport.supplementalDataElement.reference","valueReference":{"extension":[{"url":"http://hl7.org/fhir/us/davinci-deqm/StructureDefinition/extension-criteriaReference","valueString":"sde-minimal-patient"}],"reference":"Patient/jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN"}},{"url":"http://hl7.org/fhir/5.0/StructureDefinition/extension-MeasureReport.supplementalDataElement.reference","valueReference":{"extension":[{"url":"http://hl7.org/fhir/us/davinci-deqm/StructureDefinition/extension-criteriaReference","valueString":"sde-location"}],"reference":"Location/RsRPNHzwjNuhj3mxTniehTWT9iZHMJ3jAPux90m01D6PT"}},{"url":"http://hl7.org/fhir/5.0/StructureDefinition/extension-MeasureReport.supplementalDataElement.reference","valueReference":{"extension":[{"url":"http://hl7.org/

fhir/us/davinci-deqm/StructureDefinition/extension-criteriaReference","valueString":"sde-ip-
encounters"}],"reference":"Encounter/A1H45ixVFdCepcwv99pl1ebIvjJVOIcErTugl8CDMNm7w"}},{"url":"http://hl7.org/
fhir/5.0/StructureDefinition/extension-MeasureReport.supplementalDataElement.reference","valueReference":
{"extension":[{"url":"http://hl7.org/fhir/us/davinci-deqm/StructureDefinition/extension-
criteriaReference","valueString":"sde-procedure"}],"reference":"Procedure/
R0O12KSwXrdfIGU9uTiX9KGgHaPoTRM2vT4XzbNbvBEik"}}],"status":"complete","type":"individual","measure":"http://
www.cdc.gov/nhsn/fhirportal/dqm/ig/Measure/NHSNAcuteCareHospitalMonthlyInitialPopulation|1.0.0-
dev","subject":{"reference":"Patient/
jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN"},"date":"2026-04-24T14:15:51+00:00","period":
{"start":"2025-11-01T05:00:00+00:00","end":"2025-12-01T05:59:59+00:00"},"group":[{"population":[{"id":"initial-
population","extension":[{"url":"http://hl7.org/fhir/5.0/StructureDefinition/extension-
MeasureReport.population.description","valueString":"All encounters for patients of any age in an ED, observation, or
inpatient location or all encounters for patients of any age with an ED, observation, inpatient, or short stay status
during the measurement period."}],"code":{"coding":[{"system":"http://terminology.hl7.org/CodeSystem/measure-
population","code":"initial-population","display":"Initial Population"}]},"count":1}]}],"evaluatedResource":
[{"reference":"ServiceRequest/eTAnk3yEdxxBU759Ufioxn8sNcnZuHAC454Dpo7u2u88u"},{"reference":"Patient/
jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN"},{"reference":"Location/
RsRPNHzwjNuhj3mxTniehTWT9iZHMJ3jAPux90m01D6PT"},{"reference":"Procedure/
R0O12KSwXrdfIGU9uTiX9KGgHaPoTRM2vT4XzbNbvBEik"},{"reference":"Encounter/
A1H45ixVFdCepcwv99pl1ebIvjJVOIcErTugl8CDMNm7w"}]}
{"resourceType":"Patient","id":"jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN","meta":{"extension":
[{"url":"http://www.cdc.gov/nhsn/fhirportal/dqm/ig/StructureDefinition/link-received-date-
extension","valueDateTime":"2026-04-24T13:54:11Z"}],"versionId":"1","lastUpdated":"2026-01-28T19:17:08.756+00:00"
,"profile":["http://www.cdc.gov/nhsn/fhirportal/dqm/ig/StructureDefinition/cross-measure-patient"]},"extension":
[{"url":"http://hl7.org/fhir/us/core/StructureDefinition/us-core-ethnicity","extension":
[{"url":"ombCategory","valueCoding":{"system":"urn:oid:2.16.840.1.113883.6.238","code":"2135-2","display":"Hispanic
or Latino"}}]},{"url":"http://hl7.org/fhir/us/core/StructureDefinition/us-core-birthsex","valueCode":"M"},{"url":"http://
hl7.org/fhir/us/core/StructureDefinition/us-core-race","valueCoding":
{"system":"urn:oid:2.16.840.1.113883.6.238","code":"2106-3","display":"White"}}],"identifier":[{"use":"usual","type":
{"text":"CEID"},"system":"urn:oid:1.2.840.114350.1.13.93.3.7.3.688884.100","value":"XXXNHSCHACPG8YM"},
{"use":"usual","type":{"text":"EPI"},"system":"urn:oid:1.2.840.114350.1.13.93.3.7.5.737384.0","value":"E38171"},
{"use":"usual","type":{"text":"EXTERNAL"},"system":"urn:oid:1.2.840.114350.1.13.93.3.7.2.698084","value":"Z02445"},
{"use":"usual","type":{"text":"FHIR"},"system":"http://open.epic.com/FHIR/StructureDefinition/patient-dstu2-fhir-
id","value":"jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN"},{"use":"usual","type":{"text":"FHIR
STU3"},"system":"http://open.epic.com/FHIR/StructureDefinition/patient-fhir-
id","value":"eoYMSVqleH50WfN2HlgKJh40eOryKLuH6hysGq8jUoA8C"},{"use":"usual","type":
{"text":"INTERNAL"},"system":"urn:oid:1.2.840.114350.1.13.93.3.7.2.698084","value":"Z91553"},{"use":"usual","type":
{"text":"MRN"},"system":"urn:oid:2.16.840.1.113883.3.16.100.1","value":"488950597"},{"use":"usual","system":"https://
open.epic.com/FHIR/StructureDefinition/PayerMemberId","value":"2113399039"}],"active":false,"name":
[{"text":"Eric014 Daniel084 Murillo042","family":"Murillo042","given":["Eric014","Daniel084"]}],"telecom":
[{"system":"phone","value":"+1
254-555-4939"}],"gender":"female","birthDate":"2008-09-11","deceasedBoolean":false,"address":[{"use":"home","line":
["15331 Mays Plains Suite 849081"],"city":"Littlechester","state":"MO","postalCode":"72321"}],"contact":[{"name":
{"use":"usual","text":"White030, Samantha033"}}],"communication":[{"language":{"coding":
[{"system":"urn:ietf:bcp:47","code":"en","display":"English"}],"text":"English"},"preferred":true}]}
{"resourceType":"Location","id":"RsRPNHzwjNuhj3mxTniehTWT9iZHMJ3jAPux90m01D6PT","meta":
{"versionId":"1","lastUpdated":"2026-01-28T19:17:08.956+00:00","profile":["http://www.cdc.gov/nhsn/fhirportal/dqm/
ig/StructureDefinition/ach-monthly-location"]},"status":"active","name":"Pediatric Burn Ward","mode":"instance","type":
[{"coding":[{"system":"https://www.cdc.gov/nhsn/cdaportal/terminology/codesystem/
hsloc.html","code":"1078-5","display":"Pediatric Burn Ward"}]}],"telecom":[{"system":"phone","value":"+1
396-555-0921"}],"address":{"use":"work","line":["222 Nathan
Lodge043"],"city":"Alexanderton","state":"PA","postalCode":"54431"},"physicalType":{"coding":[{"system":"https://
terminology.hl7.org/6.1.0/CodeSystem-location-physical-type.html","code":"wa","display":"Ward"}]},"partOf":
{"reference":"Location/iUe9pd2zFJfSLWLFwfEmBw5TfYjOaaN9AkN6JU7IhmpRD"}}
{"resourceType":"Procedure","id":"R0O12KSwXrdfIGU9uTiX9KGgHaPoTRM2vT4XzbNbvBEik","meta":{"extension":
[{"url":"http://www.cdc.gov/nhsn/fhirportal/dqm/ig/StructureDefinition/link-received-date-
extension","valueDateTime":"2026-04-24T13:57:08Z"}],"versionId":"1","lastUpdated":"2026-01-28T19:17:11.072+00:00"
,"profile":["http://www.cdc.gov/nhsn/fhirportal/dqm/ig/StructureDefinition/ach-monthly-
procedure"]},"status":"completed","code":{"coding":[{"system":"http://snomed.info/
sct","code":"185349003","display":"Insertion of subcutaneous infusion pump"}]},"subject":{"reference":"Patient/
jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN"},"encounter":{"reference":"Encounter/
A1H45ixVFdCepcwv99pl1ebIvjJVOIcErTugl8CDMNm7w"},"performedPeriod":
{"start":"2025-11-29T01:17:51Z","end":"2025-11-29T02:07:51Z"}}
{"resourceType":"Encounter","id":"A1H45ixVFdCepcwv99pl1ebIvjJVOIcErTugl8CDMNm7w","meta":{"extension":
[{"url":"http://www.cdc.gov/nhsn/fhirportal/dqm/ig/StructureDefinition/link-received-date-

extension","valueDateTime":"2026-04-24T13:54:11Z"}],"versionId":"1","lastUpdated":"2026-01-28T19:17:08.97+00:00","profile":["http://www.cdc.gov/nhsn/fhirportal/dqm/ig/StructureDefinition/ach-monthly-encounter"]},"extension":[{"url":"http://open.epic.com/FHIR/StructureDefinition/extension/accidentrelated","valueBoolean":false},{"url":"https://www.lantanagroup.com/fhir/StructureDefinition/nhsn-encounter-original-status","valueCoding":{"code":"TRIAGED"}}],"status":"finished","class":{"system":"http://terminology.hl7.org/CodeSystem/v3-ActCode","code":"IMP","display":"inpatient encounter"},"type":[{"coding":[{"system":"http://snomed.info/sct","code":"183452005","display":"Emergency hospital admission (procedure)"}],"text":"Emergency hospital admission (procedure)"}],"serviceType":{"coding":[{"system":"http://terminology.hl7.org/CodeSystem/service-type","code":"319","display":"Diaphragm Fitting"}],"text":"Diaphragm Fitting"},"subject":{"reference":"Patient/jEY0jhwcYACCAMEZwhEmUytjgNiWXrPfueFL6kJzIqRUN","display":"Eric014 Daniel084 Murillo042"},"period":{"start":"2025-11-29T00:47:51Z","end":"2025-11-29T17:49:57Z"},"reasonCode":[{"coding":[{"system":"http://snomed.info/sct","code":"423590009","display":"Clostridium difficile colitis (disorder)"}],"text":"C. difficile colitis"},{"coding":[{"system":"urn:oid:1.2.840.114350.1.13.277.3.7.2.728286","code":"35","display":"Diarrhea"}],"text":"Diarrhea"}],"diagnosis":[{"condition":{"reference":"Condition/IGSvNitj"},"use":{"coding":[{"system":"http://terminology.hl7.org/CodeSystem/diagnosis-role","code":"AD","display":"Admission diagnosis"}]}}],"hospitalization":{"admitSource":{"coding":[{"system":"http://terminology.hl7.org/CodeSystem/admit-source","code":"psych","display":"From psychiatric hospital"}],"text":"From psychiatric hospital"},"dischargeDisposition":{"coding":[{"system":"http://terminology.hl7.org/CodeSystem/discharge-disposition","code":"psy","display":"Psychiatric hospital"}],"text":"Psychiatric hospital"}},"location":[{"location":{"reference":"Location/RsRPNHzwjNuhj3mxTniehTWT9iZHMJ3jAPux90m01D6PT","display":"Pediatric Burn Ward"},"period":{"start":"2025-11-29T09:52:54Z","end":"2025-11-29T17:49:57Z"}}]}
{"resourceType":"OperationOutcome","id":"cd751f88-e03c-4ee6-a518-37a4f5158a02","issue":[{"severity":"error","code":"invalid","diagnostics":"Patient has failed Validation"}]}

How To Associate Patient Submissions to the Manifest

Each submitted patient .ndjson file will contain a single FHIR Patient resource and at least one MeasureReport. If a user needs to cross check that a patient's submission is in the manifest or vice versa, the following can be done:

- To ensure a patient's submission is in the manifest:
  - The Patient.Id reference must be included in the manifest List.entry of the manifest List resource.
  - The MeasureReport.Id reference must be included in the manifest MeasureReport.contained.List.entry of the manifest MeasureReport resource.
- To ensure manifest entries have a corresponding patient submission:
  - Each MeasureReport.contained.List.entry must correspond to a single patient submission. A potential error in the submission process may have occurred if a MeasureReport.Id on the manifest does not coincide with any patient.ndjson Measure Reports.

Comparisons Between Link MVP and Link Cloud Submissions

MVP	Link Cloud
Patient files are in .json format	Patient files are in .ndjson format

Patient file is a FHIR Bundle resource that contains all resources used for evaluation.	Patient file has a new line delimited list of all resources used for evaluation
Patient bundles are submitted for an entire reporting period at once	Patient files are submitted immediately after evaluation and pre-qual validation is complete
Manual submission of patient bundles indicates end of reporting period	The existence of the manifest.ndjson file in the facility submission directory indicates end of reporting period
No indication that patient failed prequal validation	Patient.ndjson file and manifest.ndsjon file will contain an OperationOutcome resource indicating that there was a failed validation
Facility related resources (Medication, Location, etc) exist in the patient Bundle	Facility related resources (Medication, Location, etc) exist in the patient .ndjson files

Report Service

Type: Service

Overview

The Report service is responsible for persisting the Measure Reports and FHIR resources that the Measure Eval service generates after evaluating a patient against a measure. When a tenant's reporting period end date has been met, the Report Service performs various workflows to determine if all of the patient MeasureReports are accounted for that period prior to initiating the submission process.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Configuration

The following environment variables and settings are used for the Report Service:

Variable	Description
ASPNETCORE_ENVIRONMENT	The .NET environment (e.g., Docker, Development, Production)
ServiceInformation__ServiceName	The name of the service
ServiceInformation__ProductVersion	The product version
EnableSwagger	Enables Swagger API documentation
KafkaConnection__BootstrapServers__0	Kafka broker address
KafkaConnection__ClientId	Kafka client ID
KafkaConnection__SaslProtocolEnabled	Enables SASL protocol for Kafka
KafkaConnection__SaslUsername	Kafka SASL username

KafkaConnection__SaslPassword	Kafka SASL password
MongoDB__ConnectionString	MongoDB connection string
MongoDB__DatabaseName	MongoDB database name
MongoDB__CollectionName	MongoDB collection name
Quartz__Scheduler__InstanceName	Quartz scheduler instance name
Quartz__Scheduler__InstanceId	Quartz scheduler instance ID
Quartz__MongoDb__DatabaseName	Quartz MongoDB database name
Quartz__MongoDb__CollectionPrefix	Quartz MongoDB collection prefix
ConnectionStrings__Redis	Redis connection string
Redis__Password	Redis password
Authentication__EnableAnonymousAccess	Enables anonymous authentication
Authentication__Schemas__LinkBearer__Authority	Authority for LinkBearer authentication
Authentication__Schemas__LinkBearer__ValidateToken	Validate LinkBearer tokens
LinkTokenService__Authority	Authority for LinkTokenService
LinkTokenService__LinkAdminEmail	Admin email for LinkTokenService
LinkTokenService__TokenLifespan	Token lifespan for LinkTokenService
LinkTokenService__SigningKey	Signing key for LinkTokenService
CORS__AllowAllOrigins	Allow all origins for CORS
CORS__AllowedOrigins	List of allowed origins for CORS
CORS__AllowAllMethods	Allow all HTTP methods for CORS
CORS__AllowedMethods	List of allowed HTTP methods for CORS

CORS__AllowAllHeaders	Allow all headers for CORS
CORS__AllowedHeaders	List of allowed headers for CORS
CORS__AllowedExposedHeaders	List of exposed headers for CORS
CORS__AllowCredentials	Allow credentials for CORS
CORS__MaxAge	Max age for CORS preflight requests
Telemetry__EnableTelemetry	Enables telemetry
Telemetry__EnableRuntimeInstrumentation	Enables runtime instrumentation for telemetry
Telemetry__EnableOtelCollector	Enables OpenTelemetry collector integration
Telemetry__OtelCollectorEndpoint	OpenTelemetry collector endpoint
Telemetry__EnableAzureMonitor	Enables Azure Monitor integration
Logging__LogLevel__Default	Default log level
Logging__LogLevel__Microsoft.AspNetCore	Log level for Microsoft.AspNetCore
Logging__LogLevel__System	Log level for System
Serilog__Using__0	Serilog sink (e.g., Console, Grafana.Loki)
Serilog__MinimumLevel__Default	Serilog minimum log level
Serilog__MinimumLevel__Override__Microsoft	Serilog override for Microsoft
Serilog__MinimumLevel__Override__System	Serilog override for System
Serilog__WriteTo__0__Name	Serilog sink name (e.g., GrafanaLoki)
Serilog__WriteTo__0__Args__uri	Loki logging endpoint
Serilog__WriteTo__0__Args__labels__0__key	Serilog label key (e.g., app)

Serilog__WriteTo__0__Args__labels__0__value	Serilog label value (e.g., link-cloud)
Serilog__WriteTo__0__Args__labels__1__key	Serilog label key (e.g., component)
Serilog__WriteTo__0__Args__labels__1__value	Serilog label value (e.g., Report)
Serilog__WriteTo__0__Args__propertiesAsLabels__0	Serilog property as label (e.g., app)
Serilog__WriteTo__0__Args__propertiesAsLabels__1	Serilog property as label (e.g., component)
Serilog__WriteTo__1__Name	Serilog sink name (e.g., Console)
ServiceRegistry__TenantService__TenantServiceUrl	Tenant service URL
ServiceRegistry__TenantService__CheckIfTenantExists	Whether to check if tenant exists
ServiceRegistry__TenantService__GetTenantRelativeEndpoint	Relative endpoint for tenant service
ConsumerSettings__ConsumerRetryDuration	Retry durations for Kafka consumers
ConsumerSettings__DisableRetryConsumer	Disable retry consumer
ConsumerSettings__DisableConsumer	Disable consumer
InternalBlobStorage__ConnectionString	Connection string for internal blob storage
InternalBlobStorage__BlobContainerName	Blob container name for internal storage
AllowReflection	Allow reflection in the service
AllowedHosts	Allowed hosts for the service
ports	Exposed port for the service (default: 8072)

For a full list of environment variables and their usage, see the docker-compose.yml file and the appsettings.json file.

Common Configurations

- [Swagger]
- [Azure App Configuration]
- [Kafka Configuration]
- [Kafka Consumer Retry Configuration]
- [Service Registry Configuration]
- [CORS Configuration]
- [Token Service Configuration]
- [Service Authentication]
- [Mongo Database]

Features and Functionality

Retry Scheduling in the Report Service:

Within the Report service, retries are scheduled using Quartz.NET�s in-memory scheduler, even though the service itself leverages a persistent MongoDB-backed scheduler for critical, long-lived jobs like report period completions. The retry jobs are managed by a dedicated RetryScheduleService that uses the "InMemoryScheduler" (registered via keyed dependency injection) to schedule and execute retry operations. This approach ensures that retry jobs are lightweight, fast, and do not persist across service restarts�making them ideal for transient, short-lived retry logic where durability is not required.

Impact of InMemory vs. MongoDB Persistence:

Using the in-memory scheduler for retries means that retry jobs are lost if the service restarts or crashes, but this is often acceptable for retry logic that can be safely re-queued or recalculated. In contrast, MongoDB persistence (used for report scheduling) ensures that jobs survive restarts, support clustering, and provide fault tolerance�critical for business workflows that must not be lost. This dual approach allows the Report service to balance reliability and performance: persistent scheduling for essential workflows, and fast, resource-efficient in-memory scheduling for retries and transient operations.

Different types of report generation methods:

- Scheduled Reports - Scheduling system used to automatically create reports (aka: ScheduledReport events) based on schedule routine (ie. a CRON specification)
- Adhoc Report - Create a ScheduledReport event manually for a one-time configuration of tenant, measures and reporting period (dates). You may optionally specify the list of patients for the report; if not specified, it will ask the census service for the patients of interest during the reporting period dates.
- Regenerate Report - Provide a previously created report ID to re-generate the report, bypassing data acquisition and normalization, and skipping straight to *evaluation*. This is useful in cases where the measure has changed and a report needs to be re-evaluated against the new measure version.

Report Service

Type: Service

Overview

The Report service is responsible for persisting the Measure Reports and FHIR resources that the Measure Eval service generates after evaluating a patient against a measure. When a tenant's reporting period end date has been met, the Report Service performs various workflows to determine if all of the patient MeasureReports are accounted for that period prior to initiating the submission process.

As part of the System State Manifest enhancement, the Report Service is also responsible for pinning artifact versions at the start of the report execution and extending the **SubmitReport** command to include this information, providing the **SubmissionService** with the data needed to assemble the canonical system state.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Configuration

The following environment variables and settings are used for the Report Service:

Variable	Description
ASPNETCORE_ENVIRONMENT	The .NET environment (e.g., Docker, Development, Production)
ServiceInformation__ServiceName	The name of the service
ServiceInformation__ProductVersion	The product version
EnableSwagger	Enables Swagger API documentation
KafkaConnection__BootstrapServers__0	Kafka broker address
KafkaConnection__ClientId	Kafka client ID

KafkaConnection__SaslProtocolEnabled	Enables SASL protocol for Kafka
KafkaConnection__SaslUsername	Kafka SASL username
KafkaConnection__SaslPassword	Kafka SASL password
MongoDB__ConnectionString	MongoDB connection string
MongoDB__DatabaseName	MongoDB database name
MongoDB__CollectionName	MongoDB collection name
Quartz__Scheduler__InstanceName	Quartz scheduler instance name
Quartz__Scheduler__InstanceId	Quartz scheduler instance ID
Quartz__MongoDb__DatabaseName	Quartz MongoDB database name
Quartz__MongoDb__CollectionPrefix	Quartz MongoDB collection prefix
ConnectionStrings__Redis	Redis connection string
Redis__Password	Redis password
Authentication__EnableAnonymousAccess	Enables anonymous authentication
Authentication__Schemas__LinkBearer__Authority	Authority for LinkBearer authentication
Authentication__Schemas__LinkBearer__ValidateToken	Validate LinkBearer tokens
LinkTokenService__Authority	Authority for LinkTokenService
LinkTokenService__LinkAdminEmail	Admin email for LinkTokenService
LinkTokenService__TokenLifespan	Token lifespan for LinkTokenService
LinkTokenService__SigningKey	Signing key for LinkTokenService
CORS__AllowAllOrigins	Allow all origins for CORS
CORS__AllowedOrigins	List of allowed origins for CORS

CORS__AllowAllMethods	Allow all HTTP methods for CORS
CORS__AllowedMethods	List of allowed HTTP methods for CORS
CORS__AllowAllHeaders	Allow all headers for CORS
CORS__AllowedHeaders	List of allowed headers for CORS
CORS__AllowedExposedHeaders	List of exposed headers for CORS
CORS__AllowCredentials	Allow credentials for CORS
CORS__MaxAge	Max age for CORS preflight requests
Telemetry__EnableTelemetry	Enables telemetry
Telemetry__EnableRuntimeInstrumentation	Enables runtime instrumentation for telemetry
Telemetry__EnableOtelCollector	Enables OpenTelemetry collector integration
Telemetry__OtelCollectorEndpoint	OpenTelemetry collector endpoint
Telemetry__EnableAzureMonitor	Enables Azure Monitor integration
Logging__LogLevel__Default	Default log level
Logging__LogLevel__Microsoft.AspNetCore	Log level for Microsoft.AspNetCore
Logging__LogLevel__System	Log level for System
Serilog__Using__0	Serilog sink (e.g., Console, Grafana.Loki)
Serilog__MinimumLevel__Default	Serilog minimum log level
Serilog__MinimumLevel__Override__Microsoft	Serilog override for Microsoft
Serilog__MinimumLevel__Override__System	Serilog override for System
Serilog__WriteTo__0__Name	Serilog sink name (e.g., GrafanaLoki)

Serilog__WriteTo__0__Args__uri	Loki logging endpoint
Serilog__WriteTo__0__Args__labels__0__key	Serilog label key (e.g., app)
Serilog__WriteTo__0__Args__labels__0__value	Serilog label value (e.g., link-cloud)
Serilog__WriteTo__0__Args__labels__1__key	Serilog label key (e.g., component)
Serilog__WriteTo__0__Args__labels__1__value	Serilog label value (e.g., Report)
Serilog__WriteTo__0__Args__propertiesAsLabels__0	Serilog property as label (e.g., app)
Serilog__WriteTo__0__Args__propertiesAsLabels__1	Serilog property as label (e.g., component)
Serilog__WriteTo__1__Name	Serilog sink name (e.g., Console)
ServiceRegistry__TenantService__TenantServiceUrl	Tenant service URL
ServiceRegistry__TenantService__CheckIfTenantExists	Whether to check if tenant exists
ServiceRegistry__TenantService__GetTenantRelativeEndpoint	Relative endpoint for tenant service
ConsumerSettings__ConsumerRetryDuration	Retry durations for Kafka consumers
ConsumerSettings__DisableRetryConsumer	Disable retry consumer
ConsumerSettings__DisableConsumer	Disable consumer
InternalBlobStorage__ConnectionString	Connection string for internal blob storage
InternalBlobStorage__BlobContainerName	Blob container name for internal storage
AllowReflection	Allow reflection in the service
AllowedHosts	Allowed hosts for the service
ports	Exposed port for the service (default: 8072)

For a full list of environment variables and their usage, see the docker-compose.yml file and the

appsettings.json file.

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Kafka Consumer Retry Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- Mongo Database

Features and Functionality

Retry Scheduling in the Report Service:

Within the Report service, retries are scheduled using Quartz.NET's in-memory scheduler, even though the service itself leverages a persistent MongoDB-backed scheduler for critical, long-lived jobs like report period completions. The retry jobs are managed by a dedicated RetryScheduleService that uses the "InMemoryScheduler" (registered via keyed dependency injection) to schedule and execute retry operations. This approach ensures that retry jobs are lightweight, fast, and do not persist across service restarts—making them ideal for transient, short-lived retry logic where durability is not required.

Impact of InMemory vs. MongoDB Persistence:

Using the in-memory scheduler for retries means that retry jobs are lost if the service restarts or crashes, but this is often acceptable for retry logic that can be safely re-queued or recalculated. In contrast, MongoDB persistence (used for report scheduling) ensures that jobs survive restarts, support clustering, and provide fault tolerance—critical for business workflows that must not be lost. This dual approach allows the Report service to balance reliability and performance: persistent scheduling for essential workflows, and fast, resource-efficient in-memory scheduling for retries and transient operations.

Different types of report generation methods:

- Scheduled Reports - Scheduling system used to automatically create reports (aka: ScheduledReport events) based on schedule routine (ie. a CRON specification)
- Adhoc Report - Create a ScheduledReport event manually for a one-time configuration of tenant, measures and reporting period (dates). You may optionally specify the list of patients for the report; if not specified, it will ask the census service for the patients of interest during the reporting period dates.
- Regenerate Report - Provide a previously created report ID to re-generate the report, bypassing data acquisition and normalization, and skipping straight to *evaluation*. This is useful in cases where the measure has changed and a report needs to be re-evaluated against the new measure version.

Query Dispatch Service

Type: Service

Overview

The Query Dispatch service is primarily responsible for applying a lag period prior to making FHIR resource query requests against a facility endpoint. The current implementation of the Query Dispatch service handles how long Link Cloud should wait before querying for a patient's FHIR resources after being discharged. To ensure that the encounter related data for the patient has been settled (Medications have been closed, Labs have had their results finalized, etc), tenants are able to customize how long they would like the lag from discharge to querying to be.

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Kafka Consumer Retry Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- SQL Server Database Configuration

Normalization Service

Type: Service

Overview

FHIR resources queried from EHR endpoints can vary from location to location. There will be occasions where data for specific resources may need to be adjusted to ensure that Link Cloud properly evaluates a patient against dQM's. The Normalization service is a component in Link Cloud to help make those adjustments in an automated way. The service operates in between the resource acquisition and evaluation steps to ensure that the tenant data is in a readied state for measure evaluation.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Kafka Consumer Retry Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- SQL Server Database Configuration

Features and Functionality

The **Normalization** functionality is a critical step in preparing FHIR resources for further processing, such as measure evaluation or submission. It ensures that data adheres to expected formats, codes, and structures, enabling consistent and accurate downstream workflows.

Key Features

- Normalization is applied to **each FHIR resource** acquired from the EHR.
- The process is executed **immediately after data acquisition**.
- **Tenant-specific configuration** ensures normalization aligns with each tenant's requirements.
- When a property is normalized, an **extension is added** to the FHIR resource containing the original value for reference.

Supported Operations

Normalization is supported by the following configurable operations:

Converts codes from one system to another using a configurable list of mappings. **Use Case**: Standardizing codes between local EHR systems and external reporting requirements.

Details:
Matches a "source code/system" pair.
Converts it to the specified "target code/system."
Corrects resource IDs that are improperly formatted, such as IDs exceeding 64 characters. **Use Case**: Complying with FHIR standards and maintaining reference consistency.

Details:
Replaces the invalid ID with a **hash** of the original ID.
Updates any **references** to the corrected ID to ensure integrity.
Copies values between properties in a FHIR resource using FHIRPath expressions. **Use Case**: Populating properties required for measure evaluation or reporting.

Details:
Identifies source and destination properties using FHIRPath.
Copies the value from the source to the destination.
Description: *TODO: Document* Copies the **identifiers** from a Location resource to its **type** property. **Use Case**: Ensuring location data adheres to external standards.

Details:
Facilitates normalization of **local codes** into standardized codes via the `Location.type` field.
Ensures that the precision of `Period.start` matches the precision of `Period.end`. **Use Case**: Avoiding processing errors during measure evaluation.

Details:
Modifies `Period.start` to match the precision of `Period.end`.
Prevents errors in the CQL (measure evaluation) engine caused by inconsistent precision in Period types.

Tail Messages

Normalization propagates the end-of-stream signal by forwarding a tail **ResourceNormalized** event with acquisitionComplete = true for the same (patientId, reportTrackingId, queryType) tuple it received from acquisition. Measure Evaluation uses this tail to start evaluation for the current phase. See:

- Events → **ResourceNormalized**
- Docs → Tail Messages: Patient Completion Signals

Configuration

Normalization settings are **tenant-specific** and configured to meet the unique requirements of each tenant. This ensures flexibility and adaptability across different EHR implementations.

The order of operations in the normalization configuration is defined by the index of each operation in the dictionary/configuration.

Example:

```json
{
  "OperationSequence": {
    "1": {
      "$type": "CopyLocationIdentifierToTypeOperation"
    },
    "0": {
      "$type": "ConceptMapOperation"
    },
    "2": {
      "$type": "ConditionalTransformationOperation"
    }
  }
}
```

In this case, "ConceptMapOperation" is executed first, followed by "CopyLocationIdentifier...", followed by "ConditionalTransform...".

Notes

- Normalization ensures that all FHIR resources are processed in compliance with the applicable standards and requirements.
- Extensions added to properties retain original values, enabling transparency and traceability in the normalization process.

Measure Evaluation Service

Type: Service

Overview

The Measure Eval service is a Java based application that is primarily responsible for evaluating bundles of acquired patient resources against the measures that Link Cloud tenants are configured to evaluate with. The service utilizes the CQF framework to perform the measure evaluations.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Azure App Config
- Telemetry
- Swagger
- Mongo DB
- Kafka
- Service Authentication

Custom Configurations

Property Name	Description	Type/Value	Required	Secret?
link.reportability-predicate	Predicate to determine if a patient is reportable	"...IsInInitialPopulation" (default)	No	No

Reportability Predicates

The `link.reportability-predicate` property is used to determine if a patient is reportable. The default value is "com.lantanagroup.link.measureeval.reportability.IsInInitialPopulation", whichi s a class that implements `Predicate<MeasureReport>`. Other predicate implementations may be built over time in the same package and should be listed here.

Package `com.lantanagroup.link.measureeval.reportability`:

- IsInInitialPopulation: Determines if a patient is reportable if they are in the initial population (a count of 1 or more for the "InitialPopulation" population of the patient's MeasureReport).

Azure CosmosDB for MongoDB

The resource collection in the database needs to be created in the Azure portal and configured to shard based on the correlationId property

Features and Functionality

Measure Evaluation is a critical process in assessing clinical data against FHIR digital quality measures. It ensures that healthcare data is analyzed consistently and accurately using standardized logic and definitions.

Key Concepts

- **FHIR Digital Quality Measures**: Defined standards that outline how clinical data is measured for quality reporting and compliance.
- **Measure Package**
  : A comprehensive bundle (in FHIR JSON Bundle format) required for evaluation, including:
  - Measure definitions.
  - CQL logic in FHIR Library resources.
  - Terminology, such as pre-expanded value sets and optimized code systems.

Evaluation Process

1. **Pre-preparation**:

- Data is collected and normalized to align with FHIR standards.
- Measure packages are prepared, containing all artifacts necessary for evaluation.

2. **Execution**: Measures are executed systematically against the acquired data for each patient, including multiple evaluations during **progressive querying** as described in Progressive Querying.
3. **Results**: Each measure produces results indicating compliance or performance, which can be consumed by reporting or downstream systems. These results are in the form of a MeasureReport resource specific to the individual patient that the measure was executed against.

Role in Progressive Querying

Measure evaluation is performed multiple times for each patient during **progressive querying** to support an efficient and focused reporting pipeline:

- Determines whether the patient from the census meets the initial criteria for submission.
- Identifies what data should be submitted for the reporting scenario if the patient is relevant.
- Includes "FHIR Profile" assertion statements in the resulting data to support the validation service in

determining which profiles to validate the data against.

Tail Messages

Measure Evaluation buffers **ResourceNormalized** messages per `(patientId, reportTrackingId, queryType)` and waits for a tail where `acquisitionComplete = true`. On the tail:

- For `Initial`, it performs the initial evaluation; if reportable, it may emit **DataAcquisitionRequested** for `Supplemental`.
- For `Supplemental`, it performs the final evaluation and emits **MeasureReportGenerated**.

See:

- Events **ResourceNormalized**, **MeasureReportGenerated**
- Docs [Tail Messages: Patient Completion Signals]

Integration

Measure evaluation is often part of a broader workflow:

- **Data Acquisition**: Data is collected and normalized to a standard format.
- **Measure Execution**: Evaluations are run against pre-configured measures as data becomes available.
- **Result Propagation**: Evaluated results are consumed by the report service.

This approach ensures consistent, reliable evaluation of healthcare quality measures, supporting improved care outcomes and regulatory adherence.

Testing

The measure engine may be tested against arbitrary data using the $evaluate operation (which is custom-built for this purpose in the measure evaluation service) or using the `measureeval-cli.jar` that can be built separately from the service; see [measureeval/README.md] for more information.

Upload & Storage

- Bundles are uploaded via: `PUT /api/measureeval/measureDefinition`
- The uploaded bundle is a FHIR JSON Bundle and typically includes: `Measure`, `Library`, `StructureDefinition` (Profile), `ValueSet`, `CodeSystem`
- Bundles are stored **as-is** in the measureeval service's database.

Measure definition bundles that are generated by CQF Tooling do not include StructureDefinition resources that are necessary for validation.

Evaluation Execution

When a request is made to evaluate a measure:

1. The service checks if a `MeasureEvaluator` instance has already been compiled and cached.
2. If not, it:

- Retrieves the corresponding bundle from the database.
- Compiles a `MeasureEvaluator` instance.
- Stores the compiled instance in memory/cache for future use.

Terminology Handling:

- Currently, the bundle must include all required `ValueSet` and `CodeSystem` resources for evaluation to succeed.
- There are future plans to integrate a FHIR Terminology (TX) Service, which would offload terminology expansion (e.g., `$expand`) and eliminate the need for local terminology resources in the bundle.
- Note: At this time, value sets and code systems must be pre-expanded, or enumerate all of the codes that should be included in a value set's intensional definition. A terminology service is being explored to address this.

Known Deficiencies

- **No version tracking**: Only a single bundle per `:id` is retained.
- **Overwrites are destructive**: Uploading a new bundle overwrites the existing one.
- **No version selection**: There is no support for evaluating a measure against a specific version of the bundle.

Measure Evaluation Service

Type: Service

Overview

The Measure Eval service is a Java based application that is primarily responsible for evaluating bundles of acquired patient resources against the measures that Link Cloud tenants are configured to evaluate with. The service utilizes the CQF framework to perform the measure evaluations.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Azure App Config
- Telemetry
- Swagger
- Mongo DB
- Kafka
- Service Authentication

Custom Configurations

Property Name	Description	Type/Value	Required	Secret?
link.reportability-predicate	Predicate to determine if a patient is reportable	"...IsInInitialPopulation" (default)	No	No

Reportability Predicates

The `link.reportability-predicate` property is used to determine if a patient is reportable. The default value is "com.lantanagroup.link.measureeval.reportability.IsInInitialPopulation", whichi s a class that implements `Predicate<MeasureReport>`. Other predicate implementations may be built over time in the same package and should be listed here.

Package `com.lantanagroup.link.measureeval.reportability`:

- IsInInitialPopulation: Determines if a patient is reportable if they are in the initial population (a count of 1 or more for the "InitialPopulation" population of the patient's MeasureReport).

Features and Functionality

Measure Evaluation is a critical process in assessing clinical data against FHIR digital quality measures. It ensures that healthcare data is analyzed consistently and accurately using standardized logic and definitions.

Key Concepts

- **FHIR Digital Quality Measures**: Defined standards that outline how clinical data is measured for quality reporting and compliance.
- **Measure Package**
  : A comprehensive bundle (in FHIR JSON Bundle format) required for evaluation, including:
  - Measure definitions.
  - CQL logic in FHIR Library resources.
  - Terminology, such as pre-expanded value sets and optimized code systems.

Evaluation Process

1. **Pre-preparation**:

- Data is collected and normalized to align with FHIR standards.
- Measure packages are prepared, containing all artifacts necessary for evaluation.

2. **Execution**: Measures are executed systematically against the acquired data for each patient, including multiple evaluations during **progressive querying** as described in Progressive Querying.
3. **Results**: Each measure produces results indicating compliance or performance, which can be consumed by reporting or downstream systems. These results are in the form of a MeasureReport resource specific to the individual patient that the measure was executed against.

Role in Progressive Querying

Measure evaluation is performed multiple times for each patient during **progressive querying** to support an efficient and focused reporting pipeline:

- Determines whether the patient from the census meets the initial criteria for submission.
- Identifies what data should be submitted for the reporting scenario if the patient is relevant.
- Includes "FHIR Profile" assertion statements in the resulting data to support the validation service in determining which profiles to validate the data against.

Tail Messages

Measure Evaluation buffers **ResourceNormalized** messages per (patientId, reportTrackingId, queryType) and waits for a tail where acquisitionComplete = true. On the tail:

- For `Initial`, it performs the initial evaluation; if reportable, it may emit **DataAcquisitionRequested** for `Supplemental`.
- For `Supplemental`, it performs the final evaluation and emits **MeasureReportGenerated**.

See:

- Events **ResourceNormalized**, **MeasureReportGenerated**
- Docs [Tail Messages: Patient Completion Signals]

Integration

Measure evaluation is often part of a broader workflow:

- **Data Acquisition**: Data is collected and normalized to a standard format.
- **Measure Execution**: Evaluations are run against pre-configured measures as data becomes available.
- **Result Propagation**: Evaluated results are consumed by the report service.

This approach ensures consistent, reliable evaluation of healthcare quality measures, supporting improved care outcomes and regulatory adherence.

Testing

The measure engine may be tested against arbitrary data using the $evaluate operation (which is custom-built for this purpose in the measure evaluation service) or using the `measureeval-cli.jar` that can be built separately from the service; see [measureeval/README.md] for more information.

Upload & Storage

- Bundles are uploaded via: `PUT /api/measureeval/measureDefinition`
- The uploaded bundle is a FHIR JSON Bundle and typically includes: `Measure`, `Library`, `StructureDefinition` (Profile), `ValueSet`, `CodeSystem`
- Bundles are stored **as-is** in the measureeval service's database.

Measure definition bundles that are generated by CQF Tooling do not include StructureDefinition resources that are necessary for validation.

Evaluation Execution

When a request is made to evaluate a measure:

1. The service checks if a `MeasureEvaluator` instance has already been compiled and cached.
2. If not, it:

- Retrieves the corresponding bundle from the database.
- Compiles a `MeasureEvaluator` instance.

- Stores the compiled instance in memory/cache for future use.

Terminology Handling:

- Currently, the bundle must include all required `ValueSet` and `CodeSystem` resources for evaluation to succeed.
- There are future plans to integrate a FHIR Terminology (TX) Service, which would offload terminology expansion (e.g., `$expand`) and eliminate the need for local terminology resources in the bundle.
- Note: At this time, value sets and code systems must be pre-expanded, or enumerate all of the codes that should be included in a value set's intensional definition. A terminology service is being explored to address this.

Known Deficiencies

- **No version tracking**: Only a single bundle per `:id` is retained.
- **Overwrites are destructive**: Uploading a new bundle overwrites the existing one.
- **No version selection**: There is no support for evaluating a measure against a specific version of the bundle.

Terminology Service

Type: Service

Overview

The Terminology Service provides a focused subset of FHIR Terminology capabilities to other Link Cloud services (primarily the Validation Service):

- Hosts read and operation endpoints for ValueSet and CodeSystem resources
- Supports $validate-code on both ValueSet and CodeSystem
- Supports $expand for ValueSets, with limited support for query parameters. `filter`, `include`, `exclude`, and `inactive` are not supported.
- Publishes a CapabilityStatement at /api/terminology/fhir/metadata

On startup, the service loads configured ValueSets and CodeSystems from the local file system into an in-memory cache; runtime requests operate exclusively against this cache for predictable, fast responses and no external dependencies.

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- [Swagger]
- [Azure App Configuration]
- [Service Authentication]
- [CORS Configuration]
- [Telemetry]

Custom Configurations

Property Name	Description	Secret?
terminology.path	The path on the local file system where terminology folders and artifacts are located that are loaded into memory when the service starts	No

Notes:

- The path should contain one folder per artifact. Each folder must include a JSON file for the FHIR artifact (ValueSet or CodeSystem) and a CSV with codes to load. See repository samples for expected

formats.
- In a deployed environment, the path should be configured as an Azure App Configuration setting that is mounted as a volume to an Azure Storage Account File Share.
- Updates on disk are not auto-watched; call `POST /api/terminology/config/$reload-cache` (or restart the service) to pick up changes.

Example folder structure:

```
/data/terminology/
  /ValueSet-us-core-race/
    ValueSet-us-core-race.json
    ValueSet-us-core-race.csv
  /CodeSystem-iso-3166/
    CodeSystem-iso-3166.json
    CodeSystem-iso-3166.csv
```

Features and Functionality

- In-memory caching: all artifacts and their codes are kept in memory for fast lookups.
- ValueSet expansion: expansions are constructed from the cached codes; no external terminology server is contacted.
- Code validation: validates membership within CodeSystems/ValueSets loaded into the cache, including optional display matching.

Display Matching

1. **Case Sensitivity**: The comparison is case-sensitive

- Example: "Test Code" ≠ "test code"

2. **Culture/Locale**: Comparison is culture-invariant

- No special handling of accents or other cultural variations
- Example: "café" ≠ "cafe"

3. **Whitespace Handling**: No trimming is performed

- Leading and trailing spaces are significant
- Example: " Test" ≠ "Test"

4. **Exact Matching**: Only exact matches are accepted

- No partial or substring matches
- Example: "Test Code" ≠ "Test Code (Legacy)"

Data Acquisition Worker Service

Type: Service

Overview

The worker builds on the Data Acquisition service, which connects and queries a tenant's endpoint for FHIR resources that are needed to evaluate patients for a measure.

- When a log entry is marked ready, `DataAcquisitionLogService.StartRetrievalProcess` updates its status and produces a `ReadyToAcquire` event for the log and facility.
- `Program.cs` registers hosted services such as `ReadyToAcquireListener`, `RetryListener`, and `RetryScheduleService` to consume these events and provide retry behavior if needed.
- Inside `ReadyToAcquireListener`, each message triggers `PatientDataService.ExecuteLogRequest`, which retrieves the patient data associated with the log and facility.

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- [Azure App Configuration]
- [Kafka Configuration]
- [Kafka Consumer Retry Configuration]
- [CORS Configuration]
- [Token Service Configuration]
- [Service Authentication]
- [SQL Server Database Configuration]

Features and Functionality

The **Data Acquisition Worker** is responsible for the actual execution of FHIR queries defined by the [Data Acquisition Service].

Log Execution Flow

1. **Event Consumption**: The worker listens for `ReadyToAcquire` events. Each event contains a `LogId` and `FacilityId`.
2. **Log Claiming**: When an event is received, the worker attempts to atomically transition the log's status from `Ready` to `Queued`. This ensures that even if multiple worker instances receive the same event, only one will process it.
3. **Background Processing**: Once claimed, the work item is added to an internal `Channel`, which is processed by the `AcquisitionProcessorBackgroundService`.
4. **Resource Retrieval**
   :
   - The worker retrieves the full `DataAcquisitionLog` and its associated `QueryPlan` details.

- It executes the FHIR search queries against the tenant's endpoint.
- It handles **pagination** for large result sets.
- For **Reference** queries, it scans previously acquired resources for the specified references and fetches them.

5. **Data Normalization**: As resources are acquired, they are sent to the normalization pipeline.
6. **Completion**: Once all resources for the log are processed, the log status is updated to `Completed`, and a `ResourceAcquired` event is published.

Concurrency and Performance

The worker is designed for high-throughput data acquisition:

- **Bounded Channel**: The internal work queue has a configurable capacity (default 200) to provide backpressure.
- **Parallel Execution**: The `AcquisitionProcessorBackgroundService` uses `Parallel.ForEachAsync` with a configurable `MaxDegreeOfParallelism` (default 8) to process multiple acquisition logs concurrently.
- **Throttling**: The worker respects EHR-specific throttling limits if configured in the tenant's Query Config.

Error Handling and Retries

- **Transient Failures**: If a query fails due to a network error or EHR timeout, the worker catches the exception and updates the log status to `Pending`. The `DataAcquisitionService` will then reschedule it for retry.
- **Poison Messages**: If a message cannot be processed after multiple attempts, it is moved to a Dead Letter Queue (DLQ).
- **Logging**: All significant actions and errors are recorded in the `Notes` field of the `DataAcquisitionLog` in the database, providing a detailed audit trail for developers and support staff.

Data Acquisition Service

Type: Service

Overview

The Data Acquisition service is responsible for connecting and querying a tenant's endpoint for FHIR resources that are needed to evaluate patients for a measure. For Epic installations, Link Cloud is utilizing the Epic FHIR STU3 Patient List resource to inform which patients are currently admitted in the facility. While this is the current solution to acquiring the patient census, there are other means of patient acquisition being investigated (ADT V2, Bulk FHIR) to provide universal support across multiple EHR vendors.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Kafka Consumer Retry Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- SQL Server Database Configuration
- Data Acquisition Service Configuration

Features and Functionality

Data Acquisition is a crucial step in the report generation pipeline, responsible for obtaining clinical data from external systems, such as FHIR R4 endpoints within electronic health record (EHR) systems. By systematically acquiring and managing data, it ensures that downstream processes like evaluation and reporting are equipped with the necessary information.

Key Roles of Data Acquisition

1. **Patient List Acquisition**:

- Retrieves a FHIR List of patients from the EHR, if configured.
- Serves as the initial step for identifying patients relevant to quality measure evaluations.

2. **Individual Patient Data Acquisition**:

- Acquires detailed FHIR data for individual patients identified in the census.
- Includes all essential data elements required for initial measure evaluations.

3. **Supplemental Data Acquisition**:

- Retrieves additional data elements that may not be needed for initial evaluations but are desirable for complete submission and reporting.

Where Data Acquisition Fits in the Pipeline

Data acquisition plays a role in three distinct stages of the report generation pipeline:

1. **Patient Identification**:

- Acquiring a list of patients from the EHR to determine the cohort for evaluation.

2. **Initial Data Collection and Evaluation**:

- Obtaining the primary dataset needed for evaluating quality measures.
- Determines whether a patient qualifies for reporting based on initial measure criteria.

3. **Supplemental Data Collection**:

- Acquiring additional, non-essential data to enrich the submission.
- Completes the dataset for comprehensive evaluation and reporting.

Progressive Querying

To optimize data acquisition, the system employs a technique called **Progressive Querying**.

During progressive querying, data is acquired in stages to meet the evaluation pipeline's needs. It flows between services via Kafka topics/events, starting from data acquisition to normalization, through initial evaluation to determine patient relevance, and back to acquire supplemental data, which is then normalized and re-evaluated.

This method minimizes the data retrieved from the EHR by acquiring only what is necessary at each stage of the pipeline:

1. **Initial Querying**: Focuses on essential data needed to evaluate measures and determine patient inclusion.
2. **Supplemental Querying**: Retrieves additional data elements after initial evaluations confirm patient relevance.
3. **Final Evaluation**: Combines initial and supplemental data for comprehensive measure evaluation and reporting.

Benefits of Progressive Querying

- **Efficiency**: Reduces the volume of data retrieved from the EHR, optimizing system performance.
- **Precision**: Focuses on acquiring only data that is needed for specific stages in the pipeline.
- **Scalability**: Supports large-scale operations by limiting unnecessary data transfers.

Bulk FHIR in Data Acquisition

Bulk FHIR is a mechanism under exploration for acquiring data efficiently. However, several limitations impact its general use in data acquisition workflows:

1. **Challenges with Bulk FHIR**:

- Most implementations lack sufficient support for acquiring specific patient data.
- Filtering returned data is often not robust enough.
- To align with the goal of acquiring only necessary data, the system does not currently implement Bulk FHIR for general initial or supplemental data acquisition.

2. **Use Cases for Bulk FHIR**:

- **Patient Census Identification**:
- Bulk FHIR is a viable solution for identifying the "census of patients," analogous to using the FHIR "List" endpoint.
- It can acquire "Patient" resources for a group of patients associated with a query, filter, or registry in the EHR.
- This use case is limited to identifying patients of interest and does not address broader data acquisition.

Configuration for Data Acquisition

Data acquisition is configurable per tenant, ensuring flexibility to accommodate diverse EHR systems and data requirements. Key configurable parameters include:

- **Base FHIR URL**:
    - For general data acquisition.
    - For FHIR List (patients of interest) retrieval.
- **Authentication Information**:
    - Such as client credentials (e.g., "client id").
- **Patient Census Retrieval**:
    - FHIR List "id" or Bulk FHIR "Group ID" used for identifying the patient cohort.
- **EHR Query Throttling/Limitations**:

- Configurable settings to respect EHR query limitations (e.g., maximum queries per minute).

Configuring Query Plans

Types of query plans:

- Daily
- Weekly
- Monthly
- Discharge

"Discharge" query plans are used when a patient is discharged from the hospital. This plan is triggered by a discharge event and is used to acquire data for the patient.

All other types of query plans are used to acquire data for patients who are currently in the hospital triggered by the end *date/time* of the scheduled report. The tenant's timezone is used for this so that if the reporting period ends at 12:59:59 PM, that represents 12:59:59 PM in the tenant's timezone, not UTC time.

All times are stored in UTC format. The tenant's time zone is configured with a valid value from IANA.

Initial / Supplemental Queries

The previously mentioned progressive query phases (initial, supplemental) are configurable through the query plan. Each configured phase can contain a list of FHIR resources that must be acquired from the configured endpoint.

Query Types

For each FHIR resource that must be queried, there are two main query types that are supported:

1. Parameter: Parameters that will be appended to the FHIR search for the configured resource.
2. Reference: Any FHIR references for the configured resource found in other acquired resources for that phase will be queried for.

Example Plan

Below is an example of a monthly query plan that's configured to acquire the following resources:

1. Initial Query Phase:

Resource	Query Type	Description

Patient	N/A	Patient resources will always be queried for each configured query plan. No configuration is needed.
Encounter	Parameter	The following parameters are included in the search: patient id, period start date and period end date
Location	Reference	Any Location FHIR references found in other acquired initial resources will be queried for. 'SearchPost' will perform an HTTP POST search for Locations rather than a GET. If an OperationType is not added, it will default to performing a GET search. Link Here for more info on FHIR searches.

2. Supplemental Query Phase:

Resource	Query Type	Description
MedicationRequest	Parameter	The following parameters are included in the search: patient Id, period start date, period end date, and the literal value 'order' in the MedicationRequest.intent element.
Medication	Reference	Any Medication FHIR references found in other acquired supplemental resources will be queried for.


```
{
  "PlanName": "NHSNdQMAcuteCareHospitalInitialPopulation",
  "FacilityId": "st-marys-hospital",
  "EHRDescription": "",
  "LookBack": "P0D",
  "Type": "Monthly",
  "InitialQueries": {
    "0": {
      "ResourceType": "Encounter",
      "QueryConfigType": "Parameter",
      "Parameters": [
        {
          "ParameterType": "Variable",
          "Name": "patient",
          "Variable": 0,
          "Format": null
        },
        {
          "ParameterType": "Variable",
          "Name": "date",
          "Variable": 1,
          "Format": "ge{0}"
        },
        {
          "ParameterType": "Variable",
          "Name": "date",
          "Variable": 3,
          "Format": "le{0}"
        }
```

```
      ]
    },
    "1": {
      "QueryConfigType": "Reference",
      "ResourceType": "Location",
      "OperationType": "SearchPost",
      "Paged": 100
    }
  },
  "SupplementalQueries": {
    "0": {
      "QueryConfigType": "Parameter",
      "ResourceType": "MedicationRequest",
      "Parameters": [
        {
          "ParameterType": "Variable",
          "Name": "patient",
          "Variable": 0,
          "Format": null
        },
        {
          "ParameterType": "Variable",
          "Name": "authoredon",
          "Variable": 1,
          "Format": "ge{0}"
        },
        {
          "ParameterType": "Variable",
          "Name": "authoredon",
          "Variable": 3,
          "Format": "le{0}"
        },
        {
          "ParameterType": "Literal",
          "Name": "intent",
          "Literal": "order"
        }
      ]
    },
    "1": {
      "QueryConfigType": "Reference",
      "ResourceType": "Medication",
      "OperationType": "SearchPost",
      "Paged": 100
    }
  }
}
```

Configuring Census and Data Sources

Data sources (where the FHIR server is located and how to authenticate) are configured via "Query Configs". There is currently no association between a query *plan* and data source. Whenever data acquisition attempts to execute a query plan against a data source, it uses the FHIR server and authentication method specified by the "Query Config", for the specified facility/tenant.

TODO: Add details about how to authenticate against Epic, Cerner, Basic, and/or OAuth data sources.

Query Plans and Acquisition Logs

The Data Acquisition Service uses **Query Plans** to define the strategy for retrieving data. These plans are translated into **Data Acquisition Logs**, which represent discrete units of work to be executed by the worker service.

Acquisition Log Lifecycle

Each acquisition task follows a strictly managed state machine:

1. **Scheduled**: The log entry is created but not yet ready for execution. This usually happens when a **DataAcquisitionRequested** event is received.
2. **Ready**: The system has determined that the log is eligible for execution. A **ReadyToAcquire** event is produced for the worker.
3. **Queued**: The worker has received the **ReadyToAcquire** event and successfully "claimed" the log.
4. **InProgress**: The worker is actively querying the FHIR endpoint and processing resources.
5. **Completed**: All resources for the log have been successfully acquired and normalized.
6. **Failed**: An error occurred during acquisition that exceeded the maximum retry attempts.

Log Creation Process

When a **DataAcquisitionRequested** event is received, the service:

1. Retrieves the appropriate **Query Plan** for the facility and report type.
2. Identifies the target patients (either from the event itself or by creating a **Census** log).
3. Generates a set of `DataAcquisitionLog` entries for each required resource type and patient, assigned to the **Initial** phase.

Execution and Dependency Management

The system determines when a log should be executed based on its `Status` and `QueryPhase`:

- **Phase-Based Execution**: Logs are typically executed in phases. **Initial** phase logs are created first. Once initial data is acquired and evaluated by downstream services, they may trigger **Supplemental** acquisition by sending new **DataAcquisitionRequested** events with a supplemental phase flag.
- **Wait Logic**: Supplemental logs are not created until the initial evaluation confirms the patient's relevance to the report. This prevents unnecessary data retrieval for patients who do not meet the report's criteria.
- **Retry Mechanism**: If a log fails due to transient issues (e.g., network timeout), it is incremented and returned to a `Pending` or `Scheduled` state for retry, up to a maximum of 5 attempts.

Query Plan Structure

A Query Plan relates to logs by defining the "blueprint" for their creation:

- **ResourceType**: The FHIR resource to be queried (e.g., Patient, Encounter, Observation).
- **QueryConfigType**
  :
  - `Parameter`: Appends specific FHIR search parameters (e.g., `date=ge2024-01-01`).

    - Reference: Instructs the system to find references to this resource type within *other* already acquired resources.

- **QueryPhase**: Categorizes the query into Initial or Supplemental.

For more details on how these logs are processed, see the Data Acquisition Worker Service.

Tail Messages

Data Acquisition emits a tail **ResourceAcquired** event for each (patientId, reportTrackingId, queryType) tuple when all resources for the phase are produced. This tail sets acquisitionComplete = true and signals downstream services to proceed. See:

- Events � **ResourceAcquired**
- Docs � Tail Messages: Patient Completion Signals

Logging

The Logging configuration defines the logging levels for different parts of the application.

Property	Description	Required	Default Value	Secret?
Logging__LogLevel__Default	Default log level	No		No
Logging__LogLevel__Microsoft.AspNetCore	Log level for asp net core logs	No		No
Logging__LogLevel__System	Level for system logs	No		No

Validation Service

Type: Service

Overview

The Validation service is a Java based application that is responsible for validating FHIR resources against the FHIR specification and any additional constraints defined by the Link Cloud tenant. The service utilizes the HAPI FHIR library to perform the validation.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Azure App Config
- Telemetry
- Swagger
- SQL Server
- Kafka
- Service Authentication

Environment Variables

Env Variable	Description	Type/Value	Secret?
JAVA_TOOL_OPTIONS	Specify min/max Java heap size	-Xms1024 -Xmx2048	No

Custom Configurations

Property Name	Description	Type/Value	Secret?
artifact.init	Whether or not to initialize the artifacts in the database with default artifacts	true (default) or false	No
link.fhir-terminology-service-url	URL of an external standards-compliant FHIR Terminology server used directly for terminology	URL	No

	calls. (First priority)		
link.terminology-service-url	Base URL of the Link Terminology Service; service appends /api/terminology/fhir automatically. (Second priority)	URL	No
cache.type	Cache implementation used for validate-code calls	none, memory, redis	No
cache.validate-code.ttl	TTL for validate-code cache entries (in seconds)	integer (seconds)	No
spring.data.redis.host	Redis host (used when cache.type=redis)	string	No
spring.data.redis.port	Redis port (used when cache.type=redis)	integer	No
spring.data.redis.password	Redis password (used when cache.type=redis)	string	Yes
spring.data.redis.username	Redis username (optional; used when cache.type=redis)	string	No
spring.data.redis.database	Redis database index (used when cache.type=redis)	integer	No

Features and Functionality

New Installation Notes

After a new installation of the validation service, the following should be run/executed:

- /api/aritfact/$initialize endpoint should be run to initialize the database with artifacts that are embedded in the validation service.
- /api/category/$initialize endpoint should be run to initialize the database with default categories stored in /src/main/resources/categories.json (within the code-base)

PENDING: This functionality is going to be altered so that they are automatically initialized on service startup when no artifacts or categories already exist in the service's database.

Upload & Storage

Artifacts are uploaded either individual or as an NPM package (preferred).

- Individually: PUT /api/validation/artifact/:type/:name

- `type` = "RESOURCE"
- `name` = FHIR resource `id`
- As a package: `PUT /api/validation/artifact/:type/:name`
- `type` = "PACKAGE"
- `name` = NPM package id

Note: The Admin UI currently only supports uploading a FHIR `Bundle` of resources. Doing so implies that the Admin UI can only upload individual resources from that Bundle to the validation service. We may consider using NPM packages in the Admin UI and aligning the MeasureEval service and the Validation service to use just NPM packages.

Process Flow

- The Measure Evaluation Service evaluates a patient. - Once evaluation is complete, it produces a Kafka `ResourceEvaluated` message for each resource returned by measure evaluation (including the MeasureReport). The report service consumes `ResourceEvaluated` and persists the resources. * When the `MeasureReport` is processed by the Report service, it uses that to determine when it has received and persisted *all* resources. * After all resources are persisted, the status of the patient for the submission changes to `ReadyToValidate`. * After the status of the patient's submission changes, it produces a `ReadyForValidation` event. - The Validation Service consumes the Kafka `ReadyForValidation` event. - It retrieves the **MeasureReport** for the specified patient from the Report Service. - It extracts all contained FHIR resources and constructs a **Bundle** for validation. - Each resource in the bundle is validated individually. - The validation process includes: - **FHIR Core Specification Validation**: Ensures compliance with the base FHIR standard. - **Profile Validation**: Each resource is checked against the profiles asserted in `meta.profile`. - If a required **StructureDefinition** (profile) is missing, a **warning** is generated: *"Can't find profile http://.../us-core-observation"* - **ValueSet and CodeSystem Validation**: Ensures that coded elements conform to the expected ValueSets and CodeSystems. - If a required **ValueSet** or **CodeSystem** is missing, a **warning** is generated: *"Can't find value set XXX"* - All validation results are aggregated into a single **OperationOutcome**, capturing any validation issues. - The **OperationOutcome** containing all validation issues is stored for further processing or review. - The categorization process is initiated against the validation issues found, and each issue is matched (if possible) to a category. - Categorized results are persisted in the database. - The validation service produces a `ValidationComplete` Kafka event and includes an indication of whether the patient's submission is valid.

Configuration

The Validation Service supports two types of artifacts that define validation rules:

1. **Package (`package.tgz` format)**

- A packaged collection of FHIR artifacts (profiles, ValueSets, CodeSystems).

2. **FHIR Resource Artifacts**

- Individual **StructureDefinitions**, **ValueSets**, and **CodeSystems** can be provided.

In addition to artifacts, categories must be initialized/specified in order to have categorized results. Otherwise, all validation results end up being "uncategorized".

Validation Categories

Validation categories are used to group similar validation issues together, providing a way to manage and prioritize them. Each category defines a set of rules (matchers) that determine which validation issues belong to it.

Category Properties

Each category consists of the following properties:

- **ID**: A unique identifier for the category (e.g., `Incorrect_display_value_for_code`).
- **Title**: A human-readable name for the category.
- **Severity**: The severity level assigned to issues in this category (`ERROR`, `WARNING`, or `INFORMATION`).
- **Acceptable**
  : A boolean flag indicating if the issues in this category are considered acceptable.
  - *Note: In the future, this flag will be used to determine if a report should be submitted (e.g., if all issues are marked as acceptable, the report can still be submitted).*

- **Guidance**: Instructions or information on how to resolve the issues in this category.
- **Matcher**: A rule or set of rules used to match validation issues.

Rule Matching Fields

Rules can be keyed on the following fields of a validation result (OperationOutcome.issue):

- **MESSAGE**: Matches against the human-readable description of the issue.
- **SEVERITY**: Matches against the FHIR severity of the issue (e.g., `error`, `warning`, `information`).
- **CODE**: Matches against the FHIR issue type code (e.g., `code-invalid`, `value`).
- **EXPRESSION**: Matches against the FHIRPath expression pointing to the element in the resource that caused the issue.

Matcher Types

Validation matchers are used to define the rules for a category. All matchers support an `inverted` property, which allows for logical negation of the match result.

- **RegexMatcher**: The primary matcher used for field-level matching. It evaluates a regular expression against a specific field of a validation issue.
- **InvertibleMatcher**: This is the base abstract implementation for other matchers. While not used directly, it provides the `inverted` property to its subclasses (`RegexMatcher` and `CompositeMatcher`), allowing any rule to be negated.
- **CompositeMatcher**: A powerful matcher that allows grouping multiple other matchers (including other composite matchers) to create complex matching logic. It uses a `requiresAllChildren` flag to determine if it should act as a logical AND (true) or logical OR (false).

Detailed Matcher Examples

Below are more detailed examples showing the differences and use cases for each matcher implementation.

1. RegexMatcher

The RegexMatcher is the most straightforward way to match a validation issue. It targets a specific field of the OperationOutcome.issue.

Example: Matching a specific code

```
{
  "field": "CODE",
  "regex": "^code-invalid$"
}
```

2. InvertibleMatcher (Negation)

Since RegexMatcher and CompositeMatcher both extend InvertibleMatcher, they can both be negated using the inverted flag.

Example: Negative Match (Anything except a specific message) This will match any validation issue *except* those where the message starts with "Success".

```
{
  "field": "MESSAGE",
  "regex": "^Success",
  "inverted": true
}
```

3. CompositeMatcher (Logical Grouping)

The CompositeMatcher is used to combine multiple rules. It can be used as a logical **AND** or a logical **OR**.

Example: Logical AND (Matching a specific code AND a specific severity) Matches only if the issue has a code of value AND a severity of error.

```
{
  "requiresAllChildren": true,
  "children": [
    {
      "field": "CODE",
      "regex": "^value$"
    },
    {
      "field": "SEVERITY",
      "regex": "^error$"
    }
  ]
}
```

Example: Logical OR (Matching any of multiple fields) Matches if either the message OR the expression contains "Patient".

```
{
  "requiresAllChildren": false,
  "children": [
    {
      "field": "MESSAGE",
      "regex": "Patient"
    },
    {
      "field": "EXPRESSION",
      "regex": "Patient"
    }
  ]
}
```

Example: Complex Nested Logic Matches if the severity is error AND the issue does NOT have a code of informational.

```
{
  "requiresAllChildren": true,
  "children": [
    {
      "field": "SEVERITY",
      "regex": "^error$"
    },
    {
      "field": "CODE",
      "regex": "^informational$",
      "inverted": true
    }
  ]
}
```

Example Categories

Example 1: Incorrect Display Value (Acceptable)

This category matches issues where a code's display name is incorrect. It is marked as acceptable: true.

```
{
  "id": "Incorrect_display_value_for_code",
  "title": "Incorrect display value for code",
  "severity": "WARNING",
  "acceptable": true,
  "guidance": "The display name for the code does not match the expected value in the terminology server. This is usually a minor issue and does not affect the validity of the data itself.",
  "matcher": {
    "field": "MESSAGE",
    "regex": "^Wrong Display Name '.*' for .* should be .*'.*' .*"
  }
}
```

Example 2: Unable to Match Profile (Not Acceptable)

This category matches issues where a resource's profile cannot be found. It is marked as `acceptable: false`.

```
{
  "id": "Unable_to_match_profile",
  "title": "Unable to match profile",
  "severity": "ERROR",
  "acceptable": false,
  "guidance": "The resource asserts a profile that is not available in the validation service. Please ensure all required profiles are uploaded as artifacts.",
  "matcher": {
    "field": "MESSAGE",
    "regex": "^Unable to find a match for profile .* among choices:"
  }
}
```

The system reserves a category with an ID of "uncategorized" for any validation issues within a report that are not mapped or associated with a defined category. This ensures that all validation issues are always captured, even when they don't match any of the configured category rules.

Categories can be configured individually or in bulk via the API.

Example: Profile Validation

When the validation service encounters an Observation resource like the following:

```
{
  "resourceType": "Observation",
  "meta": {
    "profile": ["http://.../us-core-observation"]
  }
  // ... other properties
}
```

It will:

- Validate the resource against the core FHIR specification.
- Validate against the http://.../us-core-observation profile.
- If the profile is missing, generate a warning.

Similarly, for properties bound to a ValueSet or CodeSystem, the service expects these artifacts to be provided. If they are missing, it will issue warnings.

Sequence Diagram

The following diagram illustrates the relationship between the Measure Evaluation Service, Kafka, and the Validation Service:

```
sequenceDiagram
```

```
participant MES as Measure Evaluation Service
participant Kafka as Kafka
participant VS as Validation Service
participant RS as Report Service
participant TX as Terminology Service

MES->>Kafka: Publish "ResourceEvaluated" message
Kafka->>VS: Consume "ResourceEvaluated" message
VS->>RS: Request resources for patient
RS-->>VS: Return Bundle with resources for requested patient
loop For each resource in the bundle
  VS->>VS: Validate against FHIR Core Specification
  VS->>VS: Validate against meta.profile StructureDefinition
  alt Profile Not Found
    VS->>VS: Generate warning: "Can't find profile http://..."
  end
  alt Terminology service configured
    alt cache hit
      VS->>VS: Return cached validation result
    else cache miss
      VS->>TX: Validate ValueSet and CodeSystem bindings using terminology service
    end
  else
    VS->>VS: Validate ValueSet and CodeSystem bindings using built-in/common supports
  end
  alt ValueSet/CodeSystem Not Found
    VS->>VS: Generate warning: "Can't find value set XXX"
  end
end
VS->>VS: Aggregate results into OperationOutcome
VS->>Storage: Store OperationOutcome
VS->>RS: Produces ValidationComplete event
```

Terminology Service Integration and Caching

The Validation Service validates coded elements using either a remote terminology capability or local in-memory supports, depending on how the service is configured.

Terminology resolution options (in order of precedence):

- External FHIR Terminology server: If configured, the validator directs terminology operations (e.g., $validate-code) to the external server.
- Link Terminology Service: If enabled, the validator uses Link's built-in Terminology Service (FHIR endpoint) for terminology operations.
- Local supports only: If no remote option is configured, the validator falls back to built-in/common supports (e.g., common code systems and in-memory validation support). In this mode, validations that require external ValueSets/CodeSystems may produce warnings (e.g., "Can't find value set XXX").

Caching behavior:

- Remote validate-code calls are wrapped with a cache to reduce repeated network requests.
- The cache implementation is selectable (e.g., none, in-memory, or distributed) and a TTL can be set to control how long results are retained.
- The cache key is derived from the tuple (codeSystem, code, display, valueSetUrl), ensuring semantically distinct requests are cached separately.
- A distributed cache option supports shared caching across instances (e.g., via Redis), while an in-memory option keeps cache local to the process.

Configuration reference:

- For exact property names, possible values, Redis connection options, and sample YAML, see the Custom Configurations section above.
- If using the Link Terminology Service, the service automatically targets its FHIR endpoint.
- If no terminology endpoint is configured, the validator will not make network calls for terminology; only built-in/common supports are used.

Known Deficiencies

- **Global scope**: All stored artifacts are always loaded; there is no filtering or scoping based on tenant or package.
- **No tenant-specific configuration**: There is no ability to configure validation behavior per tenant or per package version.
- Use of the HAPI FHIR validation libraries only supports `CodeSystem`, `ValueSet`, and `StructureDefinition` resources.

Future Considerations

- Operation to bulk *retrieve* categories and their rules that can be updated in an text editor and then provided back to the *bulk save* operation.
- Operation to validate *and* categorize a resource (or Bundle) and return a composite response of the validation results and associated categories.
- Operation to re-validate and re-categorize a given report, to update the persisted set of results and categories for the report.

Audit Service

Type: Service

Overview

The Audit service is responsible for persisting auditable events that are generated by the Link Cloud services.

The service provides REST endpoints for managing and retrieving audit logs. These endpoints allow for the creation, retrieval, and deletion of audit records, facilitating comprehensive tracking of system activities.

[SchemaViewer omitted in PDF]

Nodes

[NodeGraph omitted in PDF]

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Kafka Consumer Retry Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- SQL Server Database Configuration

Account Service

Type: Service

Overview

The Account service is responsible for maintaining roles and permissions for Link Cloud users.

[SchemaViewer omitted in PDF]

Configuration

Property	Type	Description
Redis:Enabled	boolean	If enabled, Redis will be used to cache account information associated with authenticated users.

Nodes

[NodeGraph omitted in PDF]

- **Image Name**: link-account

Common Configurations

- Swagger
- Azure App Configuration
- Kafka Configuration
- Service Registry Configuration
- CORS Configuration
- Token Service Configuration
- Service Authentication
- SQL Server Database Configuration

Events

Validation Complete

Type: Event

Overview

Sent once validation is finished, indicating whether the submission met all validation rules.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Resource Normalized

Type: Event

Overview

Produced after the resource is transformed according to configured mappings so evaluation components can work with a consistent format.

[NodeGraph omitted in PDF]

Event Schema

Key

A JSON object (ResourceKey) containing the facility ID and correlation ID is used as the Kafka message key.

```
{
  "facilityId": "string",
  "correlationId": "string"
}
```

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Resource Normalized

Type: Event

Overview

Produced after the resource is transformed according to configured mappings so evaluation components can work with a consistent format.

[NodeGraph omitted in PDF]

Event Schema

Key

A JSON object (`ResourceKey`) containing the facility ID and correlation ID is used as the Kafka message key.

```
{
  "facilityId": "string",
  "correlationId": "string"
}
```

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Resource Evaluated

Type: Event

Overview

Generated once evaluation is complete so the evaluated resource or MeasureReport can be stored for later submission.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Resource Acquired

Type: Event

Overview

Emitted for each resource returned during acquisition and includes the resource content along with information about the request.

[NodeGraph omitted in PDF]

Event Schema

Key

A JSON object (ResourceKey) containing the facility ID and correlation ID is used as the Kafka message key.

```
{
  "facilityId": "string",
  "correlationId": "string"
}
```

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Report Status Updated

Type: Event

Overview

The ReportStatusUpdated event is a centralized mechanism for tracking the granular progress of a report. It is produced by processing services (Data Acquisition, Measure Evaluation, Validation, Submission) and consumed by the **ReportService** to update the state of reports and patients.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Report Scheduled

Type: Event

Overview

This message carries the reporting window and other configuration details so scheduling and reporting components can persist the information and begin their respective workflows.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Ready to Acquire

Type: Event

Overview

The ReadyToAcquire event is produced by the Data Acquisition Service when a data acquisition log is ready to be processed. The Data Acquisition Service sends the event to the “ReadyToAcquire” Kafka topic with the log ID as the key.

[NodeGraph omitted in PDF]

Event Schema

Key

The log ID string (log.Id) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Ready for Validation

Type: Event

Overview

Emitted after all patient resources have been stored, allowing the validation component to fetch the bundle and perform checks.

[NodeGraph omitted in PDF]

Event Schema

Key

A JSON object containing the facility ID is used as the Kafka message key.

```
{
  "facilityId": "string"
}
```

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Payload Submitted

Type: Event

Overview

Acknowledges successful creation of a patient bundle or manifest so reporting workflows can update their status.

[NodeGraph omitted in PDF]

Event Schema

Key

A JSON object containing the facility ID and report schedule ID is used as the Kafka message key.

```
{
  "facilityId": "string",
  "reportScheduleId": "string"
}
```

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Patient Lists Acquired

Type: Event

Overview

Used to update admission or discharge status and to trigger additional processing for each patient.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Patient Event

Type: Event

Overview

Marks a change in patient status so other components know when to start their waiting period before requesting data.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Patient Census Scheduled

Type: Event

Overview

Signals that the most recent census should be gathered, allowing downstream components to keep their patient lists up to date.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Measure Report Generated

Type: Event

Overview

Produced by **MeasureEvalService** after evaluating a patient against a measure. It includes metadata about the generated report and references to where the report is stored.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Measure Evaluated

Type: Event

Overview

Signals that the evaluation phase has finished for the patient in the context of the report and no further evaluation is required.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Auditable Event Occurred

Type: Event

Overview

Emitted whenever important user or system activity happens so the audit service can maintain a comprehensive log.

[NodeGraph omitted in PDF]

Event Schema

Key

The facility ID (string) is used as the Kafka message key.

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Commands

Validate ValueSet Code

Type: Command

Summary

A REST operation implemented by the Terminology service that validates a code against a value set or code system using terminology artifacts loaded in memory to support high performance.

The validation code operation complies with the FHIR Operation $validate-code on ValueSet specification.

Example Requests

Using query parameters with the ValueSet identified in the URL path

Note: The "system" query parameter often has special characters in it that need to be URL encoded, as in this example.

```
POST {{terminology-api-base}}/api/terminology/fhir/ValueSet/address-type/$validate-code?
code=physical&system=http%3A%2F%2Fhl7.org%2Ffhir%2Faddress-type
```

Using query parameters

Note: The "system" *and* "url" query parameters often have special characters that must be URL-encoded, as in this example.

```
POST {{terminology-api-base}}/api/terminology/fhir/ValueSet/$validate-code?
url=http%3A%2F%2Fhl7.org%2Ffhir%2FValueSet%2Faddress-
type&code=physical&system=http%3A%2F%2Fhl7.org%2Ffhir%2Faddress-type
```

Example Response

Good/valid response


```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "result",
      "valueBoolean": true
    }
  ]
}
```

Bad/invalid response


```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "result",
      "valueBoolean": false
    },
    {
      "name": "message",
      "valueString": "Code not found in ValueSet"
    }
  ]
}
```


[NodeGraph omitted in PDF]

Validate CodeSystem Code

Type: Command

Summary

A REST operation implemented by the Terminology service that validates a code against a code system using terminology artifacts loaded in memory to support high performance.

The validation code operation complies with the FHIR [Operation $validate-code on CodeSystem]() specification.

The implementation of this operation is almost identical to the $validate-code operation on ValueSet. The "system" parameter is not required, in this case, because the code system is explicitly identified in the URL.

Example Requests

Using query parameters with the CodeSystem identified in the path of the URL

```
POST {{terminology-api-base}}/api/terminology/fhir/CodeSystem/address-type/$validate-code?code=physical
```

Using query parameters

Note: The "url" query parameter often has special characters that must be URL-encoded, as in this example.

```
POST {{terminology-api-base}}/api/terminology/fhir/CodeSystem/$validate-code?
url=http%3A%2F%2Fhl7.org%2Ffhir%2Faddress-type&code=physical
```

Example Response

Good/valid response


```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "result",
      "valueBoolean": true
    }
  ]
}
```

Bad/invalid response


```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "result",
      "valueBoolean": false
    },
    {
      "name": "message",
      "valueString": "Code not found in code system"
    }
  ]
}
```


[NodeGraph omitted in PDF]

Submit Report

Type: Command

Overview

Sent when all patient data for a reporting period has been processed, prompting the packaging of the final submission bundle.

[NodeGraph omitted in PDF]

Command Schema

Key


```
{
  "facilityId": "string",
  "startDate": "date-time",
  "endDate": "date-time"
}
```


Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Submit Report

Type: Command

Overview

Sent when all patient data for a reporting period has been processed, prompting the packaging of the final submission bundle. This command is being extended to include pinned artifact and versioning information to support the System State Manifest.

[NodeGraph omitted in PDF]

Command Schema

Key


```
{
  "facilityId": "string",
  "startDate": "date-time",
  "endDate": "date-time"
}
```


Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Generate Report Requested

Type: Command

Overview

Contains facility and timing information used to initiate an on-demand report generation process.

[NodeGraph omitted in PDF]

Command Schema

Key

- type: string
- value: facility id

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Evaluation Requested

Type: Command

Overview

Sent when evaluation should start for a patient; includes identifiers and context so the evaluation component can process the data.

[NodeGraph omitted in PDF]

Command Schema

Key

- type: string
- value: facility id

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Data Acquisition Requested

Type: Command

Overview

Carries patient identifiers and report context, prompting the data retrieval component to query the facility for the necessary resources.

[NodeGraph omitted in PDF]

Command Schema

Key

- type: string
- value: facility id

Headers

Header Name	Description
X-Correlation-Id	The correlation ID associated with the event that should be used in logs produced during the event

Payload

[SchemaViewer omitted in PDF]

Queries

Get Service Info

Type: Query

Overview

A RESTful query made to a service to get basic identification information like name and version.

[NodeGraph omitted in PDF]

REST API Details

- **Method**: `GET`
- **Path**: `/api/facility/info` (Note: for Tenant Service specifically)

Response

Returns identification and version information for the service.

Get Facility Configuration

Type: Query

Overview

A RESTful query made to the Tenant service to obtain the full configuration of a specific facility, including its reporting schedule and time zone.

[NodeGraph omitted in PDF]

REST API Details

- **Method**: GET
- **Path**: /api/Facility/{facilityId}

Request

Name	Type	Description
facilityId	string	The unique identifier of the facility.

Response

Returns a FacilityConfig object.

[SchemaViewer omitted in PDF]

Get Facilities List

Type: Query

Overview

A RESTful query made to the Tenant service to retrieve a list of all facilities configured in the system.

[NodeGraph omitted in PDF]

REST API Details

- **Method**: GET
- **Path**: /api/Facility

Parameters

Name	Type	Description
pageNumber	integer	The page number to retrieve.
pageSize	integer	The number of records per page.

Response

Returns a paged list of facility configurations.

Extra/Custom Documentation

Development Docs

Type: Documentation

Use this folder for engineering guidance and development standards. Most docs live under patterns, but add new sections here if they represent a new development area.

Testing

Type: Documentation

Deployment Validation

1. Confirm that the following Link Cloud service containers are running:
   - link-account
   - link-audit
   - link-bff
   - link-census
   - link-dataacquisition
   - link-measureeval
   - link-normalization
   - link-notification
   - link-querydispatch
   - link-report
   - link-submission
   - link-tenant
   - link-validation

2. Confirm that the following Link Cloud dependent containers are running:
   - **Databases/Cache**
     - sql-server
     - mongo
     - redis

   - **Telemetry/Observability**
     - otel-collector
     - grafana
     - loki
     - promeutheus
     - tempo

   - **Kafka**
     - kafka-broker
     - kafka-rest-proxy
     - kafka-ui

3. Two 'init' containers are creating SQL database tables and Kafka topics. After the following containers run their process, they will automatically stop.
   - kafka-init
   - sql-init

4. Open a web browser and access Kafka UI. By default, the page can be accessed at `http://localhost:9095`. Click the `Topics` tab and ensure that Kafka topics exist (see example screenshot below). If there aren't any topics populated (shown in the image above), attempt to rerun the following command: `docker compose up kafka_init -d`
5. Open SQL Server Management Studio and connect. The default server name should be
   `127.0.0.1,1433`
   . The
   `SA`
   password can be found in the

.env
file (The
LINK_DB_PASS
variable) in the Link Cloud repository. Ensure that the following databases were created:

- link-account
- link-audit
- link-census
- link-dataacquisition
- link-normalization
- link-notification
- link-querydispatch
- link-tenant
- link-validation

[Image omitted]
If the databases listed above don't exist, you may attempt to rerun the following command: `docker compose up mssql_init -d`

Basic Testing

Configuration database scripts are provided to generate a tenant named Test-Hospital. These scripts will load most of the service configurations needed to run Link Cloud. To be capable of performing resource acquisition, a FHIR endpoint will need to be manually added (detailed below). Here are the steps to run the scripts:

1. Open SQL Server Management Studio and connect to your SQL Server Docker container (Defaulted to port 1433).

2. Open and run \link-cloud\Scripts\load-sql-data.sql.

3. To properly query for FHIR resources, update the FHIR endpoints in the SQL link-dataacquisition database:

```
UPDATE fhirQueryConfiguration
set FhirServerBaseUrl = '** INSERT FHIR BASE URL HERE **'

UPDATE fhirListConfiguration
set FhirBaseServerUrl = '** INSERT FHIR BASE URL HERE **'
```

4. Open Mongosh in a separate terminal and run the following: load("\\link-cloud\\Scripts\\load-mongo-data.js").

Manual Reporting

Automated report scheduling can be configured through the Tenant API. However, a manual approach can be done to immediately generate a report. Open Kafka UI and produce the following into the ReportScheduled topic:

Key:

```
{
 "FacilityId": "Test-Hospital",
 "ReportType": "NHSNdQMAcuteCareHospitalInitialPopulation"
}
```

Value:

```
{
 "Parameters": [
 {
  "Key": "StartDate",
  "Value": "2024-09-01T00:00:00"
 },
 {
  "Key": "EndDate",
  "Value": "2024-09-30T23:59:59"
 }]
}
```

Because the report generation process runs immediately after the set `EndDate`, allocate enough time (a few minutes) to perform the subsequent steps.

To manually admit patients, produce the following into the `PatientListsAcquired` topic in Kafka UI:

[!NOTE] The patients included in the event below are examples only. Add patient identifiers that exist in your EHR endpoint.

Key:

```
Test-Hospital
```

Value:

```
{
   "ReportTrackingId": null,
   "PatientLists": [
      {
         "ListType": "Discharge",
         "TimeFrame": "Between24To48Hours",
         "PatientIds": []
      },
      {
         "ListType": "Admit",
         "TimeFrame": "LessThan24Hours",
         "PatientIds": [
            "hJOGSZyFOTIiwEBWUU6p9NKxflMTOLCRRpLbAkfPrfqee",
            "0VmZaB90pc5yRSefoK6sW9C9WVOvPARAgquBFNtGr6LXk"
         ]
      },
      {
         "ListType": "Discharge",
         "TimeFrame": "MoreThan48Hours",
```

```
      "PatientIds": []
    },
    {
      "ListType": "Admit",
      "TimeFrame": "Between24To48Hours",
      "PatientIds": [
        "9mDh8TdqYuZP4JZkjbyDkw1SRwOG6TOhCY8GfClh5QG2m"
      ]
    },
    {
      "ListType": "Admit",
      "TimeFrame": "MoreThan48Hours",
      "PatientIds": [
      ]
    },
    {
      "ListType": "Discharge",
      "TimeFrame": "LessThan24Hours",
      "PatientIds": [
      ]
    }
  ]
}
```


To discharge a patient and begin their reporting workflow, produce the same PatientListsAcquired event without the discharged patient in the list. The example below will discharge Patient/0VmZaB90pc5yRSefoK6sW9C9WVOvPARAgquBFNtGr6LXk:

Key:

Test-Hospital

Value:


```
{
  "ReportTrackingId": null,
  "PatientLists": [
    {
      "ListType": "Discharge",
      "TimeFrame": "Between24To48Hours",
      "PatientIds": []
    },
    {
      "ListType": "Admit",
      "TimeFrame": "LessThan24Hours",
      "PatientIds": [
        "hJOGSZyFOTIiwEBWUU6p9NKxflMTOLCRRpLbAkfPrfqee"
      ]
    },
    {
      "ListType": "Discharge",
      "TimeFrame": "MoreThan48Hours",
      "PatientIds": []
    },
    {
      "ListType": "Admit",
      "TimeFrame": "Between24To48Hours",
      "PatientIds": [
        "9mDh8TdqYuZP4JZkjbyDkw1SRwOG6TOhCY8GfClh5QG2m"
      ]
```

```
    },
    {
      "ListType": "Admit",
      "TimeFrame": "MoreThan48Hours",
      "PatientIds": [
      ]
    },
    {
      "ListType": "Discharge",
      "TimeFrame": "LessThan24Hours",
      "PatientIds": [
        "0VmZaB90pc5yRSefoK6sW9C9WVOvPARAgquBFNtGr6LXk"
      ]
    }
  ]
}
```

At the end of the reporting period, the Report service will make additional requests to query and evaluate patients that are currently admitted in the facility prior to submitting. After each of those admitted patients are evaluated, the Report service will then produce a SubmitReport event to inform the Submission service that a report is complete. To access the submission package open Docker Desktop and click the link-submission container. Select the files tab and navigate to the app\submissions folder. There, you'll be able to download the submission results for the reporting period:

[Image omitted]

Reporting Event Workflow

Note: As Link Cloud continues to develop, the workflow detailed below to generate reports may be subject to large changes.

Detailed below are the steps Link Cloud takes to generate a report for a tenant reporting period. They are broken into phases. Each phase has a sequence diagram to visualize the workflow.

Report Scheduling

```
sequenceDiagram
  participant Tenant as Tenant Service
  participant Kafka
  participant Report as Report Service
  participant Query as Query Dispatch Service

  Tenant->>Kafka: Produce - ReportScheduled()
  Kafka->>Report: Consume - ReportScheduled()
  Report->>Report: Persist reporting updates()
  Report->>Report: Create job to trigger report submission for the end of the report period()
  Report->>Query: Consume - ReportScheduled()
  Query->>Query: Persist reporting updates()
```

At the beginning of a new reporting period, the Tenant service produces a ReportScheduled event. The Query Dispatch and Report services consume and persist the reporting information in the event into their databases. The Report service sets an internal cron job (based on the EndDate of the consumed event) to execute the work needed to complete the report.

Census Acquisition and Discharge

```
sequenceDiagram
  participant Census as Census Service
  participant Kafka
  participant Acquisition as Data Acquisition Service
  participant EHR
  participant Query as Query Dispatch Service

  Census->>Kafka: Produce - CensusAcquisitionScheduled()
  Kafka->>Acquisition: Consume - CensusAcquisitionScheduled()
  Acquisition->>EHR: Query STU3 Patient List()
  Acquisition-->>Kafka: Produce - PatientListsAcquired()
  Kafka->>Census: Consume - PatientListsAcquired()
  Census->>Census: Persist and admit patient ID's from the consumed list()
  Census->>Census: Discharge patients who were admitted and are no longer on the consumed list()
  Census-->>Kafka: Produce - PatientEvent(discharge)
  Kafka->>Query: Consume - PatientEvent(discharge)
  Query->>Kafka: Produce - DataAcquisitionRequested()
  Kafka->>Acquisition: Consume - DataAcquisitionRequested()
  Acquisition->>Acquisition: Schedules a lag time before querying for patient resources()
```

During the reporting period, the Census service is configured to continually request a new list of patients admitted in a facility by producing the CensusAcquisitionScheduled event. The Data Acquisition service consumed this event and queries the facility's List endpoint. After receiving a response back from the EHR endpoints, the Data Acquisition service then produces a PatientListsAcquired event that contains a list of all patients that are currently admitted in the facility.

The Census service consumes the PatientListsAcquired event and applies updates in the database for patients have been admitted or discharged.

Note: The Census service treats any patient in the PatientListsAcquired list as an admitted patient. If the Census service has a patient marked as admitted in the database, but the patient is no longer present on the consumed list, it treats the patient as a discharge.

A PatientEvent Kafka message is produced for each patient that has been discharged.

The QueryDispatch service consumes the patient events and appends the tenants' reporting information (the info consumed in the ReportScheduled event). The service then sets a cron job based on the configured lag time that the facility wants to apply for each discharge. When that lag time is met, the Query Dispatch service produces a DataAcquisitionRequested event to trigger the acquisition and evaluation steps.

Resource Acquisition and Evaluation

```
sequenceDiagram
  participant Kafka
  participant Acquisition as Data Acquisition
```

```
participant EHR
participant Normalization
participant Eval as Measure Eval
participant Report

Kafka->>Acquisition: Consume - DataAcquisitionRequested()
Acquisition->>EHR: Query for patient resources()
Acquisition-->>Kafka: Produce - ResourceAcquired()
Kafka->>Normalization: Consume - ResourceAcquired()
Normalization->>Normalization: Applies relevant normalizations (e.g., Concept Maps)
Normalization-->>Kafka: Produce - ResourceNormalized()
Kafka->>Eval: Consume - ResourceNormalized()
Eval->>Eval: Determine if all resources required to perform the evaluation()
Eval->>Eval: Evaluate()
Eval-->>Kafka: Produce - ResourceEvaluated()
Kafka->>Report: Consume - ResourceEvaluated()
Report->>Report: Persist MeasureReport's and resources()
```

A DataAcquisitionRequested event is generated for patients that have either been discharged or are still admitted when the reporting period end date is met. This event is the trigger that causes the resource acquisition, normalization and evaluation phases for a patient.

When the Data Acquisition service consumes the DataAcquisitionRequested event, the service will use the persisted query plan to make requests to the EHR endpoint for patient FHIR resources that correspond with the report type that is in the value of the consumed event. A ResourceAcquired event is produced for each resource that the EHR endpoint responds with.

The Normalization service consumes the ResourceAcquired event and applies any normalization configurations that the facility is configured for (Example: Concept Maps). After a resource has been normalized, the service produces a ResourceNormalized event.

The Measure Eval service consumes the ResourceNormalized events and persists each resource into its database. When the service has accounted for every normalized resource for a patient, it will then perform work to bundle and evaluate those resources against the CQF libraries for the report type in the event. The generated MeasureReport and its evaluated resources are then produced as separate ResourceEvaluated events

Due to the potential large size of a MeasureReport, the MeasureEval service will produce a single ResourceEvaluated event for the MeasureReport that contains only resource references. Then, the MeasureEval service will produce ResourceEvaluated events for each evaluated resource in the MeasureReport. There is information within the event that allows for the consuming Report service to associate evaluated resources to their corresponding MeasureReport.

The MeasureEval service is capable of producing DataAcquisitionRequested events if the reporting measure also includes the need for additional supplemental data for a patient that meets the initial population criteria of a measure. If this is the case, the MeasureEval service will only produce ResourceEvaluated events after it has evaluated a patient that had had their supplemental resources acquired and normalized.

When the end of the reporting period is met, the Report service will confirm that it has MeasureReport's for all discharged patients that were within the reporting period. Additionally, it will request that currently admitted patients be evaluated for that reporting period. The service does this by producing DataAcquisitionRequested events for each admitted patient which triggers the acquisition/evaluation workflow mentioned above. After those patient MeasureReports are accounted for, the Report service will produce a SubmitReport for the Submission service.

Automated Smoke Testing

An automated smoke test can be found in the `/Tests/E2ETests` folder/project which performs the following actions:

1. Loads a FHIR server with pre-defined synthetic test data
2. Loads a measure definition's evaluation artifacts in the measure eval service
3. Loads the measure definition's validation artifacts in the validation service
4. Creates a basic tenant configuration
5. Initiates the generation of an adhoc report
6. Monitors/waits for the report to complete generating
7. Downloads the report and associated data
8. Perform assertions/validation of the resulting submitted data

Tail Messages

Type: Documentation

Summary

This document explains how the platform uses a “tail message” to indicate that a patient is done being queried for a given phase, and how that signal flows across services:

- Data Acquisition Normalization Measure Evaluation
- For both Initial and Supplemental query/normalization phases

The tail message allows Measure Evaluation to know precisely when to proceed with evaluation for the patient in each phase, avoiding premature or duplicate evaluations.

What is a tail message?

A tail message is an event that marks the end of a patient’s resource stream for a specific reporting context (facility/reportTrackingId) and phase (`Initial` or `Supplemental`). It is modeled via the `acquisitionComplete` boolean carried on acquisition/normalization events:

- `ResourceAcquired.acquisitionComplete: boolean`
- `ResourceNormalized.acquisitionComplete: boolean`

When `acquisitionComplete` is `true`, it signals that, for the identified patient, no more resources for the current phase will arrive for the given report context, and downstream consumers may proceed.

Relevant schemas in this catalog:

- Events **ResourceAcquired** contains `acquisitionComplete`, `patientId`, `queryType`, `scheduledReports`, `reportableEvent`
- Events **ResourceNormalized** contains `acquisitionComplete`, `patientId`, `queryType`, `scheduledReports`, `reportableEvent`

Note: On `ResourceNormalized`, `acquisitionComplete` exists but is not a required property; when present and `true`, it functions as the tail signal.

Correlation and scope of a tail

A tail applies to the intersection of the following:

- Patient: `patientId`
- Report context: `scheduledReports[].reportTrackingId` (and associated time window `startDate`/`endDate` and `frequency`)
- Phase: `queryType` = `Initial` or `Supplemental`

Measure Evaluation correlates resources using this tuple so that multiple patients and/or reports can be processed concurrently without interference.

End-to-end flow (Initial phase)

1. Data Acquisition produces one `ResourceAcquired` per resource as it queries the EHR for the initial phase.
2. After the last initial-phase resource for the patient/report is produced, Data Acquisition emits a tail `ResourceAcquired` with `acquisitionComplete = true` (resource payload may be absent).
3. Normalization consumes `ResourceAcquired`, transforms resources, and forwards them as `ResourceNormalized` events. For the tail, Normalization propagates the end-of-stream by publishing a `ResourceNormalized` with `acquisitionComplete = true`.
4. Measure Evaluation consumes `ResourceNormalized` and buffers per patient/report/phase until it receives the tail (`acquisitionComplete = true`). It then performs the Initial evaluation for that patient/report window. If reportable, it may emit **MeasureReportGenerated** (for informational/tracking purposes) or request supplemental data.

End-to-end flow (Supplemental phase)

1. Data Acquisition executes the Supplemental plan only for patients deemed relevant (e.g., reportable) by Initial evaluation and produces `ResourceAcquired` events per resource.
2. After the last supplemental resource is produced, Data Acquisition emits a tail `ResourceAcquired` with `acquisitionComplete = true`.
3. Normalization normalizes the supplemental resources and emits `ResourceNormalized` events; for the tail, it publishes a `ResourceNormalized` with `acquisitionComplete = true`.
4. Measure Evaluation buffers supplemental resources until it receives the supplemental tail, then performs the final evaluation for the patient/report. Upon completion it emits **MeasureReportGenerated** and per-resource **ResourceEvaluated** as applicable.

Why a tail message?

- Ordering and completeness: Resource arrivals can be out-of-order and across partitions. The tail provides a definitive completion signal for the current phase.
- Determinism: Evaluations only run once the phase's resource stream is known to be complete for the patient/report tuple.
- Efficiency: Prevents unnecessary re-evaluations while resources are still in-flight.

Message and field references

- Data Acquisition Service
  - Receives: **DataAcquisitionRequested**, **PatientCensusScheduled**
  - Sends: **ReadyToAcquire**, **PatientListsAcquired**, and per-resource **ResourceAcquired** (with tail via `acquisitionComplete = true`)
  - See: Domains **DataAccess** Services **DataAcquisitionService**
- Normalization Service
  - Receives: **ResourceAcquired**
  - Sends: **ResourceNormalized** (propagates tail via `acquisitionComplete = true`)

- See: Domains **Report** Services **NormalizationService**

- Measure Evaluation Service

  - Receives: **ResourceNormalized** (Initial and Supplemental)
  - Sends: **ResourceEvaluated**, **MeasureReportGenerated**, and can request more data via **DataAcquisitionRequested** (to trigger Supplemental)
  - See: Domains Report Services MeasureEvalService

- Events

  - **ResourceAcquired** includes `acquisitionComplete`, `patientId`, `queryType`, `scheduledReports[]`, `reportableEvent`
  - **ResourceNormalized** includes `acquisitionComplete`, `patientId`, `queryType`, `scheduledReports[]`, `reportableEvent`
  - **MeasureReportGenerated**: marks completion of evaluation for a patient/report window and provides reference to the generated report.

Tail message structure examples

Initial tail (acquisition complete) carried on **ResourceAcquired**:

```
{
 "acquisitionComplete": true,
 "patientId": "12345",
 "queryType": "Initial",
 "scheduledReports": [
  {
   "reportTypes": ["NHSNdQMAcuteCareHospital"],
   "frequency": "Monthly",
   "startDate": "2026-01-01T00:00:00Z",
   "endDate": "2026-01-31T23:59:59Z",
   "reportTrackingId": "7d1f8f0e-0b1a-4db2-8e3b-7a63c4a9e4c1"
  }
 ],
 "reportableEvent": "EOM"
}
```

Propagated tail on `ResourceNormalized`:

```
{
 "acquisitionComplete": true,
 "patientId": "12345",
 "queryType": "Initial",
 "scheduledReports": [
  {
   "reportTypes": ["NHSNdQMAcuteCareHospital"],
   "frequency": "Monthly",
   "startDate": "2026-01-01T00:00:00Z",
   "endDate": "2026-01-31T23:59:59Z",
   "reportTrackingId": "7d1f8f0e-0b1a-4db2-8e3b-7a63c4a9e4c1"
  }
 ],
```

```
  "reportableEvent": "EOM"
}
```

Supplemental tails use `queryType: "Supplemental"` with the same correlation semantics.

Aggregation logic in Measure Evaluation (conceptual)

- Group incoming **ResourceNormalized** by `(patientId, reportTrackingId, queryType)`
- Buffer resources until a message with `acquisitionComplete = true` is observed
- On tail:
  - Build the evaluation bundle for the group
  - Run evaluation (Initial or Supplemental depending on `queryType`)
  - Emit **ResourceEvaluated** (per resource as applicable) and **MeasureReportGenerated** for completion
  - If Initial and the patient is reportable, emit **DataAcquisitionRequested** to trigger Supplemental

Operational considerations and edge cases

- Idempotency: Tails may be retried; consumers should treat `(patientId, reportTrackingId, queryType, acquisitionComplete=true)` as idempotent.
- Ordering: While per-partition ordering is preserved, resources and the tail can be on different partitions. Consumers must not rely on strict ordering across the entire topic—buffer/correlate instead.
- Missing tails: Monitor for stragglers/timeouts. If a tail is missing beyond an SLA window, alert or re-request acquisition.
- Duplicates: Deduplicate on a stable key (e.g., resourceId + patientId + reportTrackingId + queryType) before evaluation.
- Replays: On replay or reprocessing, tails should deterministically retrigger evaluation only if prior outputs are absent or invalidated.

Where this is implemented

These behaviors are derived from Link Cloud’s real-world implementation and the event schemas in this catalog:

- **ResourceAcquired** and **ResourceNormalized** include `acquisitionComplete` for tail signaling
- Measure Evaluation waits for the tail per patient/report/phase, then proceeds with evaluation
- Initial evaluation may request supplemental acquisition; supplemental evaluation produces final **MeasureReportGenerated**

See also:

- Domains **DataAccess** Services **DataAcquisitionService**
- Domains **Report** Services **NormalizationService**
- Domains **Report** Services **MeasureEvalService**
- Events **ResourceAcquired**, **ResourceNormalized**, **MeasureReportGenerated**

Open Telemetry

Type: Documentation

In Link we will leverage [Open Telemetry] to capture important tracing, metrics and other logging to better design and maintain distributed services.

Setting up OpenTelemetry

There is a great guide here , but to get started you will need to add the following core NuGet packages:

- https://www.nuget.org/packages/OpenTelemetry.Exporter.Console
- https://www.nuget.org/packages/OpenTelemetry.Extensions.Hosting
- https://www.nuget.org/packages/OpenTelemetry.Instrumentation.AspNetCore/1.0.0-rc9.14
- https://www.nuget.org/packages/OpenTelemetry.Exporter.OpenTelemetryProtocol

Depending on the functionality your service provides you may have interest in adding these additional packages:

- https://www.nuget.org/packages/Confluent.Kafka.Extensions.OpenTelemetry/0.2.0?_src=template
- https://www.nuget.org/packages/MongoDB.Driver.Core.Extensions.OpenTelemetry
- https://www.nuget.org/packages/OpenTelemetry.Instrumentation.Http/1.0.0-rc9.14
- https://www.nuget.org/packages/OpenTelemetry.Instrumentation.GrpcNetClient/1.5.1-beta.1
- https://www.nuget.org/packages/OpenTelemetry.Instrumentation.Runtime/1.5.0?_src=template
- https://www.nuget.org/packages/OpenTelemetry.Instrumentation.Process/

You can view additional packages here or here.

Service Configuration

Add the following configuration items to set up service name and version.

```
public class ServiceInformation
{
    public string Name { get; set; } = string.Empty;
    public string Version { get; set; } = string.Empty;
}
```

Add the following to appsettings.json:

```
"ServiceInformation": {
  "Name": "Link Audit Service",
  "Version": "1.1.0-beta"
},
```

Once these configuration settings are in place add the following to the service startup (Program.cs):

```
var serviceInformation =
builder.Configuration.GetSection(AuditConstants.AppSettingsSectionNames.ServiceInformation).Get<ServiceInform
ation>();
if (serviceInformation != null)
{
   ServiceActivitySource.Initialize(serviceInformation);
   Counters.Initialize(serviceInformation);
}
else
{
   throw new NullReferenceException("Service Information was null.");
}
```

Telemetry Configuration

Add the following configuration items to enable greater portability of your service:

```
public class TelemetryConfig
{
   public bool EnableTracing { get; set; } = true;
   public bool EnableMetrics { get; set; } = true;
   public bool EnableRuntimeInstrumentation { get; set; } = false;
   public string TraceExporterEndpoint { get; set; } = string.Empty;
   public string MetricsEndpoint { get; set; } = string.Empty;
   public string TelemetryCollectorEndpoint { get; set; } = string.Empty;
}
```

Add the following to appsettings.json:

```
"TelemetryConfig": {
  "EnableRuntimeInstrumentation": false,
  "TraceExporterEndpoint": "http://localhost:4317/",
  "MetricsEndpoint": "http://localhost:9101",
  "TelemetryCollectorEndpoint": "http://localhost:4317"
},
```

The TraceExporterEndpoint and MetricsEndpoint are only used if you want to connect directly to a Tracing service like Jaeger Tracing or a metrics scraper like prometheus.io. The TelemetryCollectorEndpoint should be what you are targeting for any moving beyond your local development environment.

Next you will need to add the TelemetryConfig to your service constants:

Once these configuration settings are in place add the following to the service startup (Program.cs) or within an IServiceCollection extension:

```
var telemetryConfig =
builder.Configuration.GetSection(NotificationConstants.AppSettingsSectionNames.Telemetry).Get<TelemetryConfig
>();
 if (telemetryConfig != null)
 {
    var otel = builder.Services.AddOpenTelemetry();

    //configure OpenTelemetry resources with application name
    otel.ConfigureResource(resource => resource
       .AddService(
```

```
            serviceName: ServiceActivitySource.Instance.Name,
            serviceVersion: ServiceActivitySource.Instance.Version
        ));

    otel.WithTracing(tracerProviderBuilder =>
            tracerProviderBuilder
                .AddSource(ServiceActivitySource.Instance.Name)
                .AddAspNetCoreInstrumentation(options =>
                    {
                        options.Filter = (httpContext) => httpContext.Request.Path != "/health"; //do not capture traces for the
health check endpoint
                    })
                .AddConfluentKafkaInstrumentation()
                .AddOtlpExporter(opts => { opts.Endpoint = new Uri(telemetryConfig.TelemetryCollectorEndpoint); }));

    otel.WithMetrics(metricsProviderBuilder =>
            metricsProviderBuilder
                .AddAspNetCoreInstrumentation()
                .AddProcessInstrumentation()
                .AddMeter("LinkAuditService")
                .AddOtlpExporter(opts => { opts.Endpoint = new Uri(telemetryConfig.TelemetryCollectorEndpoint); }));

    if(telemetryConfig.EnableRuntimeInstrumentation)
    {
        otel.WithMetrics(metricsProviderBuilder =>
            metricsProviderBuilder
                .AddRuntimeInstrumentation());
    }

    if (builder.Environment.IsDevelopment())
    {
        otel.WithTracing(tracerProviderBuilder =>
            tracerProviderBuilder
            .AddConsoleExporter());

        //metrics are very verbose, only enable console exporter if you really want to see metric details
        //otel.WithMetrics(metricsProviderBuilder =>
        //    metricsProviderBuilder
        //        .AddConsoleExporter());
    }
 }
```

The first line will need to be adjusted to retrieve your services TelemetryConfiguration

```
var telemetryConfig =
builder.Configuration.GetSection(<ServiceConstants>.AppSettingsSectionNames.Telemetry).Get<TelemetryConfig>(
);
```

You will also need to create the `ServiceActivitySource`.

DiagnosticNames

This static class ensures consistency across services when adding tags to activities, metrics, and tracing.

Creating a root activity

It is best practice to create a static root source activity that will be used to build all child activities.

```
public static class ServiceActivitySource
{
    private static string _version = string.Empty;
    public static string ServiceName = "Link Audit Service";
    public static ActivitySource Instance { get; private set; } = new ActivitySource(ServiceName, _version);

    public static void Initialize(ServiceInformation serviceInfo)
    {
        ServiceName = serviceInfo.Name;
        _version = serviceInfo.Version;
        Instance = new ActivitySource(ServiceName, _version);
    }
}
```

For more information about .NET activates see [Add distributed tracing instrumentation](#).

Creating Activities

To create an activity anywhere in your service add the following using statement:

```
using Activity? activity = ServiceActivitySource.Instance.StartActivity("Get All Audit Events Query");
```

which will cover whatever is in the current scope of the code. You can also specify the scope by

```
using Activity? activity = ServiceActivitySource.Instance.StartActivity("Get All Audit Events Query")
{
  //your code here
}
```

As you can see you are using the root source activity and creates a child activity under it.

You may also want to created nested activities, which can be done as shown in the following example:

```
using Activity? activity = ServiceActivitySource.Instance.StartActivity("List Audit Event Query");

try
{
    var (result, metadata) = await _datastore.FindAsync(searchText, filterFacilityBy, filterCorrelationBy, filterServiceBy,
filterActionBy, filterUserBy, sortBy, pageSize, pageNumber);

    //convert AuditEntity to AuditModel
    using (ServiceActivitySource.Instance.StartActivity("Map List Results"))
    {

        List<AuditModel> auditEvents = result.Select(x => new AuditModel
        {
            Id = x.Id,
            FacilityId = x.FacilityId,
            CorrelationId = x.CorrelationId,
            ServiceName = x.ServiceName,
            EventDate = x.EventDate,
            User = x.User,
```

```
            Action = x.Action,
            Resource = x.Resource,
            PropertyChanges = x.PropertyChanges?.Select(p => new PropertyChangeModel { PropertyName =
p.PropertyName, InitialPropertyValue = p.InitialPropertyValue, NewPropertyValue = p.NewPropertyValue }).ToList(),
            Notes = x.Notes
        }).ToList();

        PagedAuditModel pagedAuditEvents = new PagedAuditModel(auditEvents, metadata);

        return pagedAuditEvents;
    }
}
catch (NullReferenceException ex)
{
    _logger.LogDebug(AuditLoggingIds.ListItems, ex, "Failed to find event records.");
    var queryEx = new ApplicationException("Failed to execute the request to find audit events.", ex);
    throw queryEx;
}
```

Accessing a current activity

You can access the activity in the current scope at any time as follows:

```
//add id to current activity
var currentActivity = Activity.Current;
currentActivity?.AddTag("audit id", id);
```

In this example we are adding a tag to the current activity. For more information about tags see https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.activity.tags?view=net-7.0.

Adding Instrumentation to Kafka Producers and Consumers

If you have added the NuGet package for Kafka telemetry then you will need to make the following changes to include that instrumentation in the producer:

```
new ProducerBuilder<string,
AuditEventMessage>(_kafkaConnection.Value.CreateProducerConfig()).SetValueSerializer(new
JsonWithFhirMessageSerializer<AuditEventMessage>()).BuildWithInstrumentation();
```

Instead of just calling Build() when creating the ProducerBuilder you will need to call `BuildWithInstrumentation()`.

To add instrumentation to a consumer you need to change from `Consume()` to `ConsumeWithInstrumentation()`:

```
await _eventConsumer.ConsumeWithInstrumentation(async (result, cancellationToken) => {
  //your code here
}, cancellationToken);
```

Metrics

You can add custom metrics like counters. To do so you will want to create a service metrics class that contains all custom counters or other meters you want to keep track of in your service.

```
namespace LantanaGroup.Link.Audit.Infrastructure.Telemetry
{
    public class AuditServiceMetrics
    {
        public const string MeterName = "LinkAuditService";

        private readonly Histogram<double> _auditSearchDuration;
        private readonly TimeProvider _timeProvider;

        public AuditServiceMetrics(IMeterFactory meterFactory, TimeProvider timeProvider)
        {
            _timeProvider = timeProvider;

            Meter meter = meterFactory.Create(MeterName);
            AuditableEventCounter = meter.CreateCounter<long>("link_audit_service.auditable_event.count");
            _auditSearchDuration = meter.CreateHistogram<double>("link_audit_service.audit.search.duration", "ms");
        }

        public Counter<long> AuditableEventCounter { get; private set; }

        public TrackedRequestDuration MeasureAuditSearchDuration()
        {
            return new TrackedRequestDuration(_auditSearchDuration, _timeProvider);
        }
    }

    public class TrackedRequestDuration : IDisposable
    {
        private readonly TimeProvider _timeProvider;
        private readonly long _requestStartTime;
        private readonly Histogram<double> _histogram;

        public TrackedRequestDuration(Histogram<double> histogram, TimeProvider timeProvider)
        {
            _histogram = histogram;
            _timeProvider = timeProvider;
            _requestStartTime = timeProvider.GetTimestamp();
        }

        public void Dispose()
        {
            var elapsed = _timeProvider.GetElapsedTime(_requestStartTime);
            _histogram.Record(elapsed.TotalMilliseconds);
        }
    }
}
```

ou will then want to register the service metrics class.

```
services.AddSingleton<AuditServiceMetrics>();
```

Next you can update these counters where it makes sense in your service. Bellow you will see an `AuditableEvent` counter get incremented prior to the `CreateAuditEventCommand` returning the id of the newly created audit event. In this example the `AuditServiceMetrics` class was injected in the constructor of the

CreateAuditEventCommand.

```
public CreateAuditEventCommand(..., AuditServiceMetrics metrics)
{
  ...
  _metrics = metrics ?? throw new ArgumentNullException(nameof(metrics));
}

//Log creation of new audit event
_logger.LogAuditEventCreation(auditLog);
_metrics.AuditableEventCounter.Add(1,
    new KeyValuePair<string, object?>("service", auditLog.ServiceName),
    new KeyValuePair<string, object?>("facility", auditLog.FacilityId),
    new KeyValuePair<string, object?>("action", auditLog.Action),
    new KeyValuePair<string, object?>("resource", auditLog.Resource)
);
```

Setting up Jaeger in your local environment

To test consuming tracing you can can set up Jaeger in docker desktop as follows:

```
docker run -d --name jaeger -e COLLECTOR_OTLP_ENABLED=true -p 16686:16686 -p 4317:4317 -p 4318:4318
jaegertracing/all-in-one:latest
```

Using the configuration setup from earlier, Jaeger should now be consuming any trace data generated by open telemetry.

Here is an example of Jaeger visualizing a request to get a list of audit events.

[Image omitted]

Logging & Error Handling

Type: Documentation

Setting up SeriLog

To set up SeriLog to capture the desired logging information you will need to have the following NuGet packages installed:

- Serilog AspNetCore
  - dotnet add package Serilog.AspNetCore
- Serilog Loki
  - dotnet add package Serilog.Sinks.Grafana.Loki
- Serilog Exceptions
  - dotnet add package Serilog.Exceptions
- Serilog Encrichers Span
  - dotnet add package Serilog.Enrichers.Span
- Serilog Expressions
  - dotnet add package Serilog.Expressions

To configure SeriLog, in the RegisterServices method in the service startup (Program.cs) add the following:

```
// Logging using Serilog
builder.Logging.AddSerilog();
var loggerOptions = new ConfigurationReaderOptions { SectionName =
AuditConstants.AppSettingsSectionNames.Serilog };
Log.Logger = new LoggerConfiguration()
            .ReadFrom.Configuration(builder.Configuration, loggerOptions)
            .Filter.ByExcluding("RequestPath like '/health%'")
            .Enrich.WithExceptionDetails()
            .Enrich.FromLogContext()
            .Enrich.WithSpan()
            .Enrich.With<ActivityEnricher>()
            .CreateLogger();

//Serilog.Debugging.SelfLog.Enable(Console.Error);
```

You can add additional filters to exclude certain request paths you don’t want captured in logging (swagger for example).

Next add the following to appsettings.json

```
"Link:Audit:Logging:Serilog": {
  "Using": [ "Serilog.Sinks.Console", "Serilog.Sinks.Grafana.Loki" ],
  "MinimumLevel": {
```

```
    "Default": "Information",
    "Override": {
      "Microsoft": "Warning",
      "System": "Warning"
    }
  },
  "WriteTo": [
    { "Name": "Console" },
    {
      "Name": "GrafanaLoki",
      "Args": {
        "uri": "http://localhost:3100",
        "labels": [
          {
            "key": "app",
            "value": "Link-BoTW"
          },
          {
            "key": "component",
            "value": "Audit"
          }
        ],
        "propertiesAsLabels": [ "app", "component" ]
      }
    }
  ]
}
```

Additionally you can add the following to enhance the format of the console:

```
"Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "System": "Warning"
    },
    "Console": {
      "FormatterName": "json",
      "FormatterOptions": {
        "SingleLine": true,
        "IncludeScopes": true,
        "TimestampFormat": "HH:mm:ss ",
        "UseUtcTimestamp": true,
        "JsonWriterOptions": {
          "Indented": true
        }
      }
    }
  }
```

Adding Problem Details

Problem details is an attempt at an industry standard to carry machine readable details of errors in a HTTP response to avoid the need to define new error response formats for HTTP APIs.

[RFC-7807]

This can be accomplished by using the .NET [problem details service].

To enable to sending of Problem Details add the following in the RegisterServices method in the service startup (Program.cs):

```
//Add problem details
builder.Services.AddProblemDetails(options => {
    options.CustomizeProblemDetails = ctx =>
    {
        ctx.ProblemDetails.Detail = "An error occured in our API. Please use the trace id when requesting assistence.";
        if (!ctx.ProblemDetails.Extensions.ContainsKey("traceId"))
        {
            string? traceId = Activity.Current?.Id ?? ctx.HttpContext.TraceIdentifier;
            ctx.ProblemDetails.Extensions.Add(new KeyValuePair<string, object?>("traceId", traceId));
        }

        if (builder.Environment.IsDevelopment())
        {
            ctx.ProblemDetails.Extensions.Add("service", "Audit");
        }
        else
        {
            ctx.ProblemDetails.Extensions.Remove("exception");
        }

    };
});
```

Add the following to the SetupMiddleware method in the service startup (Program.cs):

```
if (app.Environment.IsDevelopment() || app.Environment.EnvironmentName.Equals("Local",
StringComparison.InvariantCultureIgnoreCase))
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler();
}
```

This will consume any 500 error returned by the API and turn it into a ProblemDetails model that is returned to the user. It will appear as follows:

```
{
    "type": "https://httpstatuses.io/500",
    "title": "Internal Server Error",
    "status": 500,
    "detail": "An error occured in our API. Please use the trace id when requesting assistence.",
    "traceId": "00-09f4ab47f959288745f3ef337ac773b7-fac718cf545d0a9d-01"
}
```

We can use the Trace Id to find relevant logs pertaining to the issue.

For this to work correctly you will need to return a 500 response type along with the exception in your API.

```
catch (Exception ex)
{
    _logger.LogError(new EventId(LoggingIds.GetItem, "Get Audit Event"), ex, "An exception occurred while attempting
```

```
to retrieve an audit event with an id of {id}", id);
    throw;
}
```

Logging Ids

The AuditLoggingIds.GetItem in the above _logger.LogError method is a new option in .NET that allows you to assign a Logging Id to log types. In the example above it will give the log an id of 1002 from the following static class.

```
public static class AuditLoggingIds
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;
    public const int GetItemNotFound = 1006;
    public const int UpdateItemNotFound = 1007;
    public const int EventConsumer = 2000;
    public const int EventProducer = 2001;
    public const int HealthCheck = 9000;
}
```

This can allow for finding specific log types by the id and will show up in logs as follows:

[Image omitted]

UserScope Middleware

The LantanaGroup.Link.Shared.Application.Middleware middleware has been added to the shared project. This middleware will capture the username as well as the id of the user if an authenticated user is present in the .NET HttpContext.User.Identity object. This object will be populated based on if/how you have authentication enabled. For more information about enabled authentication see TBD.

This scoped user information will automatically be added to any logs created. To add the middleware simply add the following in the SetupMiddleware method in the service startup (Program.cs):

```
app.UseRouting();
app.UseCors("CorsPolicy");
app.UseAuthentication();
app.UseMiddleware<UserScopeMiddleware>();
app.UseAuthorization();
app.UseEndpoints(endpoints => endpoints.MapControllers());
```

This middlware should come after app.UseAuthentication(). When this is added the following will be included in a scope within your logs.

[Image omitted]

Adding data to exceptions

For additional context, you may add some additional data about the issue that caused the exception to be thrown as follows:

```
catch (NullReferenceException ex)
{
   _logger.LogDebug(new EventId(LoggingIds.GetItem, "Get Audit Event"), ex, "Failed to get audit event with an id of {id}.", id);
   var queryEx = new ApplicationException("Failed to get audit event", ex);
   queryEx.Data.Add("Id", id);
   throw queryEx;
}
```

The `exception.Data.Add()` method will add this information in the following locations:

[Image omitted]

Additional information about .NET logging can be found here.

LoggerMessage Attribute

Introduced in .NET 6, the LoggerMessage allows for performant logging.

```
public static partial class Logging
{
   [LoggerMessage(
     AuditLoggingIds.GenerateItems,
     LogLevel.Information,
     "New audit event created")]
   public static partial void LogAuditEventCreation(this ILogger logger,
     [LogProperties]AuditEntity auditEvent);
}
```

The `LogProperties` attribute, requires .NET 8, will include all of the properties of the object in your log.

[Image omitted]

Kafka Consumer Error Handling

When consuming an event that cannot be deserialized, the consumer should catch the error through a ConsumeException. In the catch, the service should do the following:

- Create an audit event to trace that an error occurred.
- Produce an error event to the Error topic. Storing improperly structured events in an separate topic will allow for further investigate

- Commit the consumer result to acknowledge back to the Kafka broker that the consumed event was processed.

```
flowchart LR
  SourceTopic["SOURCE TOPIC"] --> KafkaApp["KAFKA<br>APP"]
  KafkaApp -->|1| TargetTopic["TARGET TOPIC"]
  KafkaApp -->|2| ErrorTopic["ERROR TOPIC"]

  classDef source fill:#e6f4fa,stroke:#aaa;
  classDef target fill:#e6f4fa,stroke:#aaa;
  classDef error fill:#f8d7da,stroke:#c00,color:#000;

  class SourceTopic source;
  class TargetTopic target;
  class ErrorTopic error;
```

Example:

```
ConsumeResult<ReportScheduledKey, ReportScheduledValue> consumeResult;
try
{
   consumeResult = _reportScheduledConsumer.Consume(cancellationToken);
}
catch (ConsumeException e)
{
   _logger.LogError($"Consume failure, potentially schema related: {e.Error.Reason}");
   var potentialFacilityId = Encoding.UTF8.GetString(e.ConsumerRecord.Message.Key);
   var potentialValue = Encoding.UTF8.GetString(e.ConsumerRecord.Message.Value);

   ProduceErrorEvent(potentialFacilityId, potentialValue);

   var auditValue = new AuditEventMessage
   {
      FacilityId = potentialFacilityId,
      Action = AuditEventType.Query,
      ServiceName = "QueryDispatch",
      EventDate = DateTime.UtcNow,
      Notes = $"Kafka ReportScheduled consume failure, potentially schema related \nException Message: {e.Error}",
   };

   ProduceAuditEvent(auditValue, e.ConsumerRecord.Message.Headers);

   _reportScheduledConsumer.Commit();

   continue;
}

private void ProduceErrorEvent(string key, string value) {
  var config = new ProducerConfig()
  {
     ClientId = "Error-QueryDispatch-ReportScheduled"
  };

  using (var producer = _errorProducerFactory.CreateProducer(config))
  {
     var headers = new Headers
     {
        new Header("X-Consumer", Encoding.UTF8.GetBytes("QueryDispatch")),
        new Header("X-Topic", Encoding.UTF8.GetBytes("ReportScheduledEvent"))
     };
```

```
    producer.Produce("Error", new Message<string, string>
    {
        Key = key,
        Value = value,
        Headers = headers
    });

    producer.Flush();
  }
}
```

Kafka Consumer Retry Handling

When a problem occurs while processing a consumed event (Facility not properly configured, API connection could not be established, could not connect to database, etc), we will want to attempt to perform retries to ensure that BotW processes each event successfully. When an exception is caught during the processing of an event, the service should do the following:

- Create an audit event to trace that an error occurred.
- Produce a retry event to the services Retry topic.
- Commit the consumer result to acknowledge back to the Kafka broker that the consumed event was processed.

```
flowchart LR
  ReportScheduled --> QueryDispatch
  QueryDispatch --> Error
  QueryDispatch --> Retry
  QueryDispatch --> ConsumeSuccess
  Retry --> QueryDispatchRetry
  QueryDispatchRetry --> Retry
  QueryDispatchRetry --> ConsumeSuccess
  ConsumeSuccess:::final -->|Success or Retry| Retry

  classDef final fill=#d4af37,stroke=#333,color=white;
```

Development Patterns

Type: Documentation

This folder captures reusable patterns for building Link services, including API guidance, auth policies, error handling, logging, observability, and testing practices.

Add new docs here when a pattern should be shared across teams or services.

Kafka DeadLetter/Retry Scenarios

Type: Documentation

Error Retry and Dead-letter Scenarios

This document outlines the error retry and dead-letter handling strategies for services in the platform. Each service section details its approach to Kafka message processing failures, including retry logic and dead-letter queue (DLQ) handling.

Tenant Service

The Tenant service is responsible for producing the `ReportScheduled` Kafka message. It does not directly implement retry or dead-letter handling for this message. Instead, these actions are performed by downstream services that consume the message (such as Report, QueryDispatch, etc.).

- **Message Production:** Tenant produces the `ReportScheduled` event for downstream processing.
- **Error Handling:** Retry and dead-letter logic for this message is managed by the consumer services, not by Tenant itself.
- **See Also:** Refer to the relevant consumer service sections for details on how retry and dead-letter scenarios are handled for `ReportScheduled` and other messages.

Census Service

The Census service interacts with several Kafka topics as both a producer and consumer. Below are the error retry and dead-letter handling strategies for each relevant topic.

PatientListsAcquired (Consumer)

- **Consumption:** Census consumes `PatientListsAcquired` events to process patient lists for facilities.
- **Retry:**
  - If message processing fails with a transient error, the message is retried according to the configured retry policy (using the shared `TransientExceptionHandler`).
  - **Exceptions triggering retry:** Any exception of type `TransientException` (e.g., temporary network issues, database timeouts, or other recoverable errors) will trigger the retry workflow.
  - Retry topics follow the `{TopicName}-Retry` naming convention.

- **Dead-letter:**
  - If all retries fail or a non-retryable error occurs, the message is sent to the `PatientListsAcquired-Error` dead-letter topic (using the shared `DeadLetterExceptionHandler`).
  - **Exceptions triggering dead-letter:** Any exception of type `DeadLetterException` (e.g., validation errors, unrecoverable business logic errors) or unhandled exceptions not marked as transient.

- DLQ topics follow the {TopicName}-Error naming convention.

PatientEvent (Producer)

- **Production:** Census produces PatientEvent messages to signal patient-related events (admit, discharge, etc.).
- **Retry:**
  - If message production fails, the error is logged and will enter retry topic for production failures in Census itself.
- **Dead-letter:**
  - If retry fails after 5 attempts, the message will be deadlettered.

PatientCensusScheduled (Producer)

- **Production:** Census produces PatientCensusScheduled messages to schedule census-related jobs.
- **Retry:**
  - Production errors are logged, but there is no built-in retry topic for production failures in Census.
- **Dead-letter:**
  - There is no built-in DLQ for production failures in Census. Downstream consumers handle dead-letter scenarios.

Data Acquisition Service

The Data Acquisition service manages FHIR queries and utilizes a state-managed background worker for execution.

DataAcquisitionRequested (Consumer)

- **Consumption:** Consumes DataAcquisitionRequested events to initiate data retrieval logs.
- **Dead-letter:**
  - Exceptions triggering dead-letter: Throws ArgumentNullException (triggering the DeadLetterExceptionHandler) if the CorrelationId is missing from headers.

PatientCensusScheduled (Consumer)

- **Consumption:** Consumes PatientCensusScheduled events to trigger patient list updates.
- **Dead-letter:**
  - Exceptions triggering dead-letter: Throws DeadLetterException if the FacilityId is missing from the

message key or if an error occurs during log creation.

ReadyToAcquire (Consumer - Worker Logic)

- **Consumption:** The Acquisition Worker consumes ReadyToAcquire messages to execute FHIR queries.
- **Retry Strategy (State-Based):**
  - ProcessingDelayException: If the execution date is in the future, the log status remains Pending to be picked up later.
  - ProduceException: Production failures for ResourceAcquired messages trigger a TransientException for Kafka-level retry.

- **Dead-letter:**
  - Kafka DLQ: Thrown if the LogId or FacilityId is null or empty in the message.
  - Internal Failure: If a general exception occurs during acquisition, the worker updates the DataAcquisitionLog status to Failed and records the error notes before throwing a DeadLetterException.

Producers

- **ResourceAcquired (Producer):** Produces results of successful FHIR queries.
- **Error Handling:** Failures update the DataAcquisitionLog status to Failed. Producer failures are primarily managed via the worker's internal state logic.

Normalization Service

The Normalization service processes FHIR resources acquired by the platform to ensure they meet standard evaluation requirements.

ResourceAcquired (Consumer)

- **Consumption:** Consumes ResourceAcquired messages containing FHIR resources or acquisition completion signals.
- **Retry:**
  - If a transient error occurs during processing, the message is retried via the TransientExceptionHandler.
  - **Exceptions triggering retry:** Any exception not explicitly caught as a dead-letter or transient exception type will default to the retry workflow.
  - Retry topics follow the ResourceAcquired-Retry naming convention.

- **Dead-letter:**
  - Messages are moved to the DLQ if a non-retryable error occurs or if metadata extraction fails.
  - **Exceptions triggering dead-letter:**
    - Missing FacilityId (message key).
    - Missing or empty CorrelationId in the message headers.
    - Unsupported resource types during deserialization.

    - Validation failures, such as a null message value or missing ReportableEvent.
  - DLQ topics follow the ResourceAcquired-Error naming convention.

ResourceNormalized (Producer)

- **Production:** Produces ResourceNormalized messages after successful transformation.
- **Error Handling:** If production fails, the error is logged. There is no built-in Kafka-level retry for producer failures; errors are surfaced to the background service.

Report Service

The Report service consumes and produces several Kafka topics. Below are the error retry and dead-letter handling strategies for each relevant topic, based on the actual code implementation.

ReportScheduled (Consumer)

- **Consumption:** Report consumes ReportScheduled events to schedule reports for facilities.
- **Retry:**
  - If message processing fails with a transient error (e.g., temporary network issues, database timeouts), the message is retried using the shared TransientExceptionHandler.
  - **Exceptions triggering retry:** Any exception of type TransientException or general exceptions not explicitly marked as dead-letter.
  - Retry topics follow the ReportScheduled-Retry naming convention.

- **Dead-letter:**
  - If a non-retryable error occurs (e.g., validation errors, duplicate report IDs), the message is sent to the ReportScheduled-Error dead-letter topic using the shared DeadLetterExceptionHandler.
  - **Exceptions triggering dead-letter:** Any exception of type DeadLetterException or unhandled validation/business logic errors.
  - DLQ topics follow the ReportScheduled-Error naming convention.

PatientListsAcquired (Consumer)

- **Consumption:** Report consumes PatientListsAcquired events to update patient submission entries for scheduled reports.
- **Retry:**
  - Transient errors (e.g., missing scheduled reports, temporary DB issues) trigger the retry workflow via TransientExceptionHandler.
  - Retry topics follow the PatientListsAcquired-Retry naming convention.

- **Dead-letter:**
  - Non-retryable errors (e.g., invalid patient list data) are sent to the PatientListsAcquired-Error dead-letter topic via DeadLetterExceptionHandler.

- DLQ topics follow the PatientListsAcquired-Error naming convention.

GenerateReportRequested (Consumer)

- **Consumption:** Report consumes GenerateReportRequested events to generate or regenerate reports.
- **Retry:**
  - Transient errors (e.g., missing report schedule, temporary failures) trigger the retry workflow via TransientExceptionHandler.
  - Retry topics follow the GenerateReportRequested-Retry naming convention.

- **Dead-letter:**
  - Non-retryable errors (e.g., invalid facility ID, missing required fields) are sent to the GenerateReportRequested-Error dead-letter topic via DeadLetterExceptionHandler.
  - DLQ topics follow the GenerateReportRequested-Error naming convention.

PayloadSubmitted (Consumer)

- **Consumption:** Report consumes PayloadSubmitted events to update submission status for reports and patients.
- **Retry:**
  - Transient errors (e.g., DB update failures, timeouts) trigger the retry workflow via TransientExceptionHandler.
  - Retry topics follow the PayloadSubmitted-Retry naming convention.

- **Dead-letter:**
  - Non-retryable errors (e.g., invalid payload data) are sent to the PayloadSubmitted-Error dead-letter topic via DeadLetterExceptionHandler.
  - DLQ topics follow the PayloadSubmitted-Error naming convention.

ResourceEvaluated (Consumer)

- **Consumption:** Report consumes ResourceEvaluated events to process evaluated resources and update submission entries.
- **Retry:**
  - Transient errors (e.g., missing report schedule, deserialization issues, timeouts) trigger the retry workflow via TransientExceptionHandler.
  - Retry topics follow the ResourceEvaluated-Retry naming convention.

- **Dead-letter:**
  - Non-retryable errors (e.g., missing correlation ID, invalid resource data) are sent to the ResourceEvaluated-Error dead-letter topic via DeadLetterExceptionHandler.
  - DLQ topics follow the ResourceEvaluated-Error naming convention.

Producers

The Report service produces several Kafka topics. Below are the error handling and dead-letter strategies for each relevant producer, based on the actual code implementation.

- **DataAcquisitionRequested (Producer):**
  - Produces `DataAcquisitionRequested` messages for each patient to trigger data acquisition.
  - **Error Handling:**
    - If message production fails (e.g., Kafka unavailable), an exception is thrown and logged. There is no built-in retry or DLQ for production failures; failures are surfaced to the caller and may be retried by the job or service logic.
- **ReadyForValidation (Producer):**
  - Produces `ReadyForValidation` messages when patient data is ready for validation.
  - **Error Handling:**
    - If message production fails, an exception is thrown and logged. There is no built-in retry or DLQ for production failures; failures are surfaced to the caller and may be retried by the job or service logic.
- **SubmitPayload (Producer):**
  - Produces `SubmitPayload` messages to signal that a report or patient payload is ready for submission.
  - **Error Handling:**
    - If message production fails, an exception is thrown and logged. There is no built-in retry or DLQ for production failures; failures are surfaced to the caller and may be retried by the job or service logic.
- **ReportManifest (Producer):**
  - Produces report manifest payloads and triggers `SubmitPayload` production.
  - **Error Handling:**
    - If manifest upload to blob storage fails, an audit event is produced and the failure is logged. If producing the `SubmitPayload` message fails, the error is surfaced as above.
- **AuditableEventOccurred (Producer):**
  - Produces audit event messages for operational and error tracking.
  - **Error Handling:**
    - If message production fails, the error is logged. There is no retry or DLQ for audit event production failures.

Note:

- The Report service does not implement Kafka-level retry or dead-letter topics for producer failures. All error handling for producers is at the application/service level, with errors logged and surfaced to the caller for possible retry by the job or workflow logic.

Authorization Policies

Type: Documentation

Link will enforce developer constructed authorization policies to ensure that users have the proper access required before performing any requests. This will help prevent users from having the ability to perform CRUD operations on service data or configurations without having the necessary permissions set by the facility administrators to do so. Authorization policies will be enforced both at the API Gateway and at each individual service.

Request Authorization Sequence

```
sequenceDiagram
    participant User
    participant APIGateway as API Gateway
    participant LinkAuth as Link Authorization
    participant Account
    participant Service as Requested Service (Ex: Tenant)

    %% Step 1
    User->>APIGateway: [1] Sams Authenticated Request()

    %% Step 2
    APIGateway->>LinkAuth: [2] Request Attribute(s): Authorize Policy: bool

    %% Step 3
    LinkAuth->>Account: [3] UserHasAccess(email, facilityId, group, role): bool
    Account-->>LinkAuth: bool

    %% Step 4
    LinkAuth-->>APIGateway: [4] Not authorized: Return error()

    %% Optional authorization passes:
    LinkAuth-->>APIGateway: Authorized

    %% Step 5
    APIGateway->>Service: [5] Authorized: Perform service request()

    %% Step 6
    Service->>LinkAuth: [6] Request Attribute: PolicyAuthorize(): bool
    LinkAuth->>Account: UserHasAccess(email, facilityId, group, role): bool
    Account-->>LinkAuth: bool
    LinkAuth-->>Service: Authorized

    %% Step 7
    Service-->>APIGateway: [7] Service Response()

    %% Step 8
    APIGateway-->>User: [8] Request Response()
```

1. User performs initial request. Authentication will have been completed at this point and the request will contain a JWT Token. The API Gateway has an endpoint for the request that contains authorization policies.
2. The policy extracts the email address of the user from the JWT Token and performs a request to the Account services' UserHasAccess endpoint. This Account endpoint requires that
   all
   policies pass the following parameters to determine if they have the proper permissions for the request:
   1. Email
   2. Facility Id

    3. Account Group
    4. Account Role

3. The Account service returns a bool value on whether the user is authorized for the request.
4. If the response returns a false, the API Gateway should return an error code (Potentially 401 Unauthorized).
5. If the response returns a true, the API Gateway will make the request to the service.
6. As a safety precaution, the service will perform the same authorization policy checks that the API Gateway endpoint did (Step 3).
7. The service will then send a response based on the request of the service.
8. The API Gateway will return the response back to the user.

Authorization Library

All authorization policies will be located under the Link.Authorization project in the .NET solution (AuthorizationPolicy.cs):

```
using Microsoft.AspNetCore.Authorization;

namespace Link.Authorization
{
  public static class AuthorizationPolicies
  {
    public static AuthorizationPolicy CanViewAuditLogs()
    {
      //TODO: Add Account Authorization Requirement functionality to support Account UserHasAccess()
      return new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .RequireRole("LinkAdministrator")
        .Build();
    }

    public static AuthorizationPolicy CanCreateNotifiactionConfigurations()
    {
      return new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .RequireRole("LinkAdministrator")
        .Build();
    }
  }
}
```

A policy must be registered by the service in Program.cs in order to be used.Below is an example of policies that have been registered for use in the API Gateway project:

```
  builder.Services.AddAuthorization(authorizationOptions => {
    authorizationOptions.AddPolicy("UserCanViewAuditLogs", AuthorizationPolicies.CanViewAuditLogs());
    authorizationOptions.AddPolicy("CanCreateNotifiactionConfigurations",
AuthorizationPolicies.CanCreateNotifiactionConfigurations());
    authorizationOptions.AddPolicy("CanUpdateNotifiactionConfigurations",
AuthorizationPolicies.CanUpdateNotifiactionConfigurations());
    authorizationOptions.AddPolicy("CanDeleteNotifiactionConfigurations",
AuthorizationPolicies.CanDeleteNotifiactionConfigurations());
    authorizationOptions.AddPolicy("ClientApplicationCanRead", policyBuilder => {
      policyBuilder.RequireScope("demogatewayapi.read");
    });
    authorizationOptions.AddPolicy("ClientApplicationCanCreate", policyBuilder => {
      policyBuilder.RequireScope("demogatewayapi.write");
```

```
  });
  authorizationOptions.AddPolicy("ClientApplicationCanDelete", policyBuilder => {
    policyBuilder.RequireScope("demogatewayapi.delete");
  });
```

Once a Policy has been registered to the service, it can then be enforced by adding the following attribute to the request endpoint:

```
[HttpGet]
[Authorize(Policy = "UserCanViewAuditLogs")]
[Authorize(Policy = "ClientApplicationCanRead")]
public async Task <ActionResult<PagedAuditModel>> ListAuditEvents()
{
  //code
}
```

As shown in the example, multiple Policies can be applied to a single request.

Application Configuration Management Process

Type: Documentation

Overview

To ensure services can be reliably deployed and operated across multiple environments, we maintain a central registry of required configuration settings in /app-config.json. This file provides a clear inventory of what must be provisioned for the system to function correctly.

When to Add a Setting to /app-config.json

Developers MUST add a configuration key to /app-config.json if:

1. **The setting is required** for the service to start or perform its core functions.
2. **The setting is environment-specific** (e.g., URLs, connection strings, API keys) and does not have a "one-size-fits-all" default.
3. **The setting is an override** of a default value that is expected to change in most production or production-like environments.
4. **The setting is security-sensitive** (e.g., Auth Authority, Secret Keys). Note: only the *key* and *description* are added; never the *value*.

When to Exclude a Setting

It is acceptable to exclude a setting if:

1. **It has a sufficient default value** that is unlikely to require an override in any deployed environment (e.g., internal cache timeouts, logging levels).
2. **The setting is strictly for local development** and has no relevance in deployed environments.

Integration into Workflow

Pull Requests (PRs)

- Any PR that introduces a new required configuration setting MUST include an update to /app-config.json.
- PR reviewers should verify that the description for any new setting is clear and that no sensitive values or environment-specific defaults are included.
- **Automated Check**: CodeRabbit is configured to scan for newly introduced configuration keys in appsettings.json, application.yml, and environment variable definitions. If new keys are detected without corresponding updates to /app-config.json, the PR will be flagged for correction and should be blocked until reconciled.

DevOps & Release Notes

- /app-config.json serves as the primary hand-off artifact for DevOps during environment provisioning.

- Release notes should highlight changes to `/app-config.json` to ensure transparency across teams.

File Format and Schema

The `/app-config.json` follows a structured schema:

`global` Array

Contains settings shared across most or all services (e.g., Kafka connection, Database provider).

`services` Object

Contains service-specific settings, keyed by the service name.

Configuration Entry Fields:

- `key`: The dot-delimited or colon-delimited path to the configuration setting as it appears in `appsettings.json` or environment variables.
- `description`: A brief explanation of the setting's purpose and its impact on the system.
- `required`: (Optional, default: true) A boolean indicating if the setting is strictly required for the service to function.

Security Warning

NEVER commit environment-specific values, passwords, or secrets to `/app-config.json`. This file is intended for documentation and schema enforcement only. Use secure secret management (e.g., Azure Key Vault) for actual values in deployed environments.

API Guidance

Type: Documentation

Our APIs should be built with the Representational State Transfer (REST) architectural style using HTTP as the application protocol.

Each API should strive to:

- **Be Platform Independent**: Allow any client to leverage the API through standard protocols
- **Be Persistence Agnostic**: The persistence layer can be changed without any changes to the API.
- **Evolve Independently from Clients**: Versioning of the API to allow clients to continue to use the API in a previous state
- **Resource Based**: Every API should be designed around a resource (model representation of an object)

API Standardization

Swagger

Each API should implement swagger documentation.

- .NET
- Java

Versioning

Do we want to do this?

By versioning our APIs we can make breaking changes without impacting clients. When a breaking change occurs a new version of the API can be released and clients can choose when to migrate to the new version of the API and make changes necessary to work with the new version.

URL versioning can be accomplished as follows:

- `https://tenant-service/api/v1/facilities`
- `https://tenant-service/api/v2/facilities`

Error Response

When an error occurs in an API an HTTP response with a status code of 500 (Internal Server Error) should be returned. The body of the response should be that of a the Problem Details object. See more here.

```
{
```

```
  "type": "<https://tools.ietf.org/html/rfc7231#section-6.6.1>",
  "title": "An error occurred while processing your request.",
  "status": 500,
  "detail": "An error occured in our API. Please use the trace id when requesting assistence.",
  "traceId": "00-09f4ab47f959288745f3ef337ac773b7-fac718cf545d0a9d-01"
}
```

Other common error HTTP status codes include:

- 400 Bad Request - This means that client-side input fails validation.
- 401 Unauthorized - This means the user is not authorized to access a resource. It usually returns when the user isn't authenticated.
- 403 Forbidden - This means the user is authenticated, but it's not allowed to access a resource.
- 404 Not Found - This indicates that a resource is not found.
- 500 Internal server error - This is a generic server error. It probably shouldn't be thrown explicitly.
- 502 Bad Gateway - This indicates an invalid response from an upstream server.
- 503 Service Unavailable - This indicates that something unexpected happened on server side (It can be anything like server overload, some parts of the system failed, etc.).

Hypertext as the Engine of Application State (HATEOAS)

This concept can provide a client with additional operations that can be performed against resources associated with the request. An additional benefit to this is the API can make changes to its URI scheme without breaking clients.

Since we are using swagger, I don't believe it is really necessary to include HATEOAS and more than likely it doesn't make sense to use both at the same time.

Model for hypermedia links based on [RFC-8288].

WebLink Model

```
{
  "rel": "resource",
  "href": "<URI of the action>",
  "action": "<HTTP Verb>",
  "types": ["<media types>"]
}
```

HTTP Verbs

[POST]

When a POST method will create a new resource, it will return an HTTP response with a status code of 201 (Created). The URI of the new resource will be included in the Location header of the response. The response body will contain the the APIs representation of the resource (only return the properties of the

entity that has been deemed necessary for clients, exclude sensitive or internal only properties).

When a POST method does some processing, but does not create a new resource, it will return a status code of 200 and return the result of the operation in the response body. If there is no result to return that is deemed of value to the client, the endpoint can return a status code of 204 (No Content).

If a client submits a POST request with invalid data, the endpoint should return a HTTP response with a status code of 400 (Bad Request). The response body should contain additional details as to why the request was invalid.

Example Request: https://tenant-service/api/facilities

[POST] https://tenant-service/api/facilities HTTP/1.1

Content-Type: application/json; charset=utf-8

{
  "facilityId": "FacilityID",
  "facilityName": "Facility Full Name"
}

Example Response

{
  "id": "8f9f1e36-a969-41e0-9c8f-fbb4496b0201",
  "facilityId": "FacilityID",
  "facilityName": "Facility Full Name"
}

[PUT]

An endpoint that accepts a PUT request for a resource should check to see if the resource provided by the client already exists. If the resource already exists it should update that resource, however if the resource provided by the client does not exist, it should return a Problem result and not Create an entry.

If the request contains a resource that does not exist, the format of the id will also need to be checked. If the format of the id provided does not match the UUID format that has been decided on for the resource then the API should return an HTTP response with a status code of 400 (Bad Request). The response body should contain a message that states “Invalid Id format”.

When a PUT request updates an existing resource, it will return an HTTP response with a status code of 202 (Accepted).

The request body should be a resource with an id element that has an identical value to the id in the URL. If no id element is provided, or the id does not match with the id in the URL, the endoint will return with a status code of 400 (Bad Request).

Example Request: https://tenant-service/api/facilities/{{id}}

PUT https://tenant-service/api/facilities/deba542d-0c75-449e-bea4-30671a9d7d9e HTTP/1.1

Content-Type: application/json; charset=utf-8

{
  "id": "deba542d-0c75-449e-bea4-30671a9d7d9e",
  "facilityId": "FacilityID",
  "facilityName": "Facility Full Name"
}

[PATCH]

TBD if use cases exist for this

[GET]

When a GET request successfully retrieves the resource asked for, it will return an HTTP status code of 200 (OK). The response body will contain the the APIs representation of the resource (only return the properties of the entity that has been deemed necessary for clients, exclude sensitive or internal only properties).

If the resource requested cannot be found, the response should be an HTTP status code for 404 (Not Found).

An interesting security discussion probably should happen around the Ids used. If the resource is sensitive, account for example, it is a good idea to use a UUID instead of something like an incremented identity column. Even string based identifiers can be problematic.
The reason for this is with incremented identifiers and string based names a user can make fairly good guesses of other Ids they can put int he request. This can be used for malicious intent as responses like Not Found or Unauthorized is useful information. If I get a Unauthorized I know that Id exists for instance. Then you can use that information to try other methods such as social engineering.

Example Request: https://tenant-service/api/facilities/{{id}}

GET https://tenant-service/api/facilities/deba542d-0c75-449e-bea4-30671a9d7d9e HTTP/1.1

Content-Type: application/json; charset=utf-8

Example Response

{

```
    "id": "deba542d-0c75-449e-bea4-30671a9d7d9e",
    "facilityId": "FacilityID",
    "facilityName": "Facility Full Name",
}
```

If the request is made to an endpoint that performs a search, it will return an HTTP status code of 200 (OK). The response body will contain a PagedListModel that contains an array of the resource model and a PagedMetadata object.

If the request is made to an endpoint that performs a search and no results were found, it should return an HTTP status code of 204 (No Content).

PagedModel Interface

```
public interface IPagedModel<T> where T : class
{
    public List<T> Records { get; set; }
    public PaginationMetadata Metadata { get; set; }
}
```

PaginationMetadata Interface

```
public interface IPaginationMetadata
{
    int PageSize { get; }
    int PageNumber { get; }
    long TotalCount { get; }
}
```

Example of a concrete implementation in the Demo API Gateway

```
public class PaginationMetadata : IPaginationMetadata
{
    public int PageSize { get; set; }
    public int PageNumber { get; set; }
    public long TotalCount { get; set; }
    public long TotalPages { get; set; }
    public PaginationMetadata() { }

    public PaginationMetadata(int pageSize, int pageNumber, long totalCount)
    {
        PageSize = pageSize;
        PageNumber = pageNumber;
        TotalCount = totalCount;
        TotalPages = (int)Math.Ceiling(totalCount / (double)pageSize);
    }
}
```

Example Request: https://tenant-service/api/facilities?pageSize=10&pageNumber=1

GET https://tenant-service/api/facilities?pageSize=10&pageNumber=1 HTTP/1.1

Content-Type: application/json; charset=utf-8

Example Response


```
{
  "records": [{
    "id": "1029c167-4a9b-42f5-9f8b-dc82eb301a3f",
    "facilityId": "FacilityId1",
    "facilityName": "Facility 1 Full Name",
  },
  {
    "id": "7240fc1b-fce9-4d88-ab98-90e4fb1715e2",
    "facilityId": "FacilityId2",
    "facilityName": "Facility 2 Full Name",
  }],
  "metadata": {
    "pageSize": 10,
    "pageNumber": 1,
    "totalCount": 2,
    "totalPages": 1
  }
}
```


How do we want to handle UI concerns such as populating drop down lists with key value pairs? Do we want to create custom endpoints that return only the needed key value pair. Do we want to use a GET baseUrl/resource that will return a subset of resource data and then the UI would map it to a key value pair model?

DELETE

When a DELETE request successfully removes a resource, it will return an HTTP response with a status code of 204 (No Content).

Example Request: https://tenant-service/api/facilities/{{id}}

DELETE https://tenant-service/api/facilities/deba542d-0c75-449e-bea4-30671a9d7d9e HTTP/1.1

Content-Type: application/json; charset=utf-8

HTTP Clients

Typed clients can be created for each underlying service the API will be communicating with. This use of the HttpClientFactory helps prevent build up of underlying network sockets that are not immediately released with a HttpClient is disposed. See more here.

Typed Client Interface

```
namespace LantanaGroup.Link.DemoApiGateway.Services.Client
{
    public interface INotificationService
    {
        Task<HttpResponseMessage> ListNotifications(string? searchText, string? filterFacilityBy, string? 
filterNotificationTypeBy, DateTime? createdOnStart, DateTime? createdOnEnd, DateTime? sentOnStart, DateTime? 
sentOnEnd, string? sortBy, int pageSize = 10, int pageNumber = 1);
        Task<HttpResponseMessage> CreateNotification(NotificationMessage model);
        Task<HttpResponseMessage> ListConfigurations(string? searchText, string? filterFacilityBy, string? sortBy, int 
pageSize = 10, int pageNumber = 1);
        Task<HttpResponseMessage> CreateNotificationConfiguration(NotificationConfigurationModel model);
        Task<HttpResponseMessage> UpdateNotificationConfiguration(NotificationConfigurationModel model);
        Task<HttpResponseMessage> DeleteNotificationConfiguration(Guid id);
    }
}
```

Typed Client Implementation

```
namespace LantanaGroup.Link.DemoApiGateway.Services.Client
{
    public class NotificationService : INotificationService
    {
        private readonly HttpClient _httpClient;
        private readonly IOptions<GatewayConfig> _gatewayConfig;

        public NotificationService(HttpClient httpClient, IOptions<GatewayConfig> gatewayConfig)
        {
            _httpClient = httpClient ?? throw new ArgumentNullException(nameof(httpClient));
            _gatewayConfig = gatewayConfig ?? throw new ArgumentNullException(nameof(\_gatewayConfig));
        }

        public async Task<HttpResponseMessage> CreateNotification(NotificationMessage model)
        {
            _httpClient.BaseAddress = new Uri(_gatewayConfig.Value.NotificationServiceApiUrl);
            _httpClient.DefaultRequestHeaders.Accept.Clear();
            _httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

            // HTTP GET
            HttpResponseMessage response = await _httpClient.PostAsJsonAsync($"api/notification", model);
            return response;
        }
    }
}
```

Adding typed services

```
builder.Services.AddHeaderPropagation(opts => {
    opts.Headers.Add("Authorization");
});

builder.Services.AddHttpClient<INotificationService, NotificationService>();
```

Header Propagation

Header propagation allows you to add a header value from an incoming request to a request made by the receiving API. The most likely example of this would be an authorization header.

Adding the typed client and Header Propagation

NuGet package

[NuGet package]

```
dotnet add package Microsoft.AspNetCore.HeaderPropagation --version 7.0.8
```

Adding header propagation to services

```
builder.Services.AddHeaderPropagation(opts => {
    opts.Headers.Add("Authorization");
});

builder.Services.AddHttpClient<INotificationService, NotificationService>().AddHeaderPropagation();
```

Header propagation will automatically add specified headers to the http client connections to underlying services. In this example we are propagating the Authorization header that contains the bearer token from the client request. See more [here].

Using the typed clients

```
namespace LantanaGroup.Link.DemoApiGateway.Controllers
{
    [Route("api/notification")]
    [ApiController]
    public class NotificationGatewayController : ControllerBase
    {
        private readonly ILogger<NotificationGatewayController> _logger;
        private readonly INotificationService _notificationService;
        private int maxPageSize = 20;

        public NotificationGatewayController(ILogger<NotificationGatewayController> logger, INotificationService
notificationService)
        {
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
            _notificationService = notificationService ?? throw new ArgumentNullException(nameof(notificationService));
        }

        [HttpGet]
        public async Task<ActionResult<PagedNotificationModel> ListNotifications(string? searchText, string?
filterFacilityBy, string? filterNotificationTypeBy, DateTime? createdOnStart, DateTime? createdOnEnd, DateTime?
sentOnStart, DateTime? sentOnEnd, string? sortBy, int pageSize = 10, int pageNumber = 1)
        {
            //TODO check for authorization
            try
            {
                //make sure page size does not exceed the max page size allowed
                if (pageSize > maxPageSize) { pageSize = maxPageSize; }
                if (pageNumber < 1) { pageNumber = 1; }

                //Get list of audit events using supplied filters and pagination
```

```
            HttpResponseMessage response = await _notificationService.ListNotifications(searchText, filterFacilityBy,
filterNotificationTypeBy, createdOnStart, createdOnEnd, sentOnStart, sentOnEnd, sortBy, pageSize, pageNumber);

            if (response.IsSuccessStatusCode)
            {

                var notificationList = await response.Content.ReadFromJsonAsync<PagedNotificationModel>();

                if (notificationList == null) { return NotFound(); }

                //add X-Pagination header for machine-readable pagination metadata
                Response.Headers.Add("X-Pagination", JsonSerializer.Serialize(notificationList.Metadata));
                return Ok(notificationList);
            }
            else
            {
                return NotFound();
            }
        }
        catch (Exception ex)
        {
            _logger.LogError(new EventId(LoggingIds.ListItems, "List Notifiations"), ex, "An exception occurred while
attempting to retrieve notifications.");
            return StatusCode(500, ex);
        }
    }
  }
}
```

Route Guidelines

1. Remove component name from route

To simplify route definitions, the URL within the request should be a sufficient indicator on which Link Cloud component is being accessed.

Incorrect	Correct
/api/census/config	/config
/api/data/{facilityId}/{queryPlanType}	/{facilityId}/{queryPlanType}

External requests to the Link Admin service will still need to contain the service as part of the request route in order for the service to able to redirect the request to the correct service.

[Image omitted]

For example, the Census service route to update a facility config is:

External request: /api/census/configs/facilities/{facilityId}

Internal service route: /configs/facilities/{facilityId}

The following Link Admin (BFF) appSettings includes the following to create this route proxy:

```
"route3": {
  "ClusterId": "CensusService",
  "AuthorizationPolicy": "AuthenticatedUser",
  "Match": {
    "Path": "api/census/{**catch-all}"
  },
  "Transforms": [
    { "PathPattern": "/{**catch-all}" }
  ]
},
```

2. Plural nouns

Unless they are singleton resources, nouns in the API route should be plural.

Incorrect	Correct
/api/census/config/{facilityId}	/api/configs/facilities/{facilityId}

3. Lower case route

API routes are case-sensitive. Because of this, routes should be lower case to remove any confusion.

Incorrect	Correct
/api/ReportConfig/Create	/api/reportconfig/create

4. Hyphenate to separate words rather than using spaces/camel case/pascal case

This practice improves readability and ensures compatibility across different systems and platforms.

Incorrect	Correct

/api/data/{facilityId}/fhirQueryConfiguration	/api/data/{facilityId}/fhir-query-configuration

Additional Resources

- Authorization Policies
- Logging and Error Handling
- Open Telemetry

Design Docs

Type: Documentation

This folder contains high-level design materials for Link, including architecture references and design proposals for new or changed capabilities.

- Architecture: Core system design, cross-cutting concerns, and reference diagrams.
- Proposals: Drafts and approved proposals for significant changes.

NHSN Reporting Use Case

Type: Documentation

Overview

The National Healthcare Safety Network (NHSN) reporting use case leverages the entire Link stack of services to generate automated reports for NHSN on behalf of healthcare facilities.

The primary goals of this implementation are:

- **Increasing the reliability of near-real-time reporting:** By automating the data flow and analysis, the platform ensures that reports are accurate and delivered promptly.
- **Reducing the reporting burden on facilities:** Automating the data collection and processing significantly reduces the manual effort required by healthcare staff, allowing them to focus on patient care.

Primary Reporting Workflow

The reporting workflow involves multiple facilities integrating through several stages, often split into initial and supplemental phases to ensure data completeness and accuracy.

```
graph TD
    subgraph Initial Phase
        A1[Data Acquisition] --> N1[Normalization]
        N1 --> E1[Measure Evaluation]
    end

    subgraph Supplemental Phase
        A2[Supplemental Data Acquisition] --> N2[Normalization]
        N2 --> E2[Measure Evaluation]
    end

    E1 -.-> A2
    E2 --> V[Validation]
    V --> S[Submission]

    style Initial Phase fill:#f9f,stroke:#333,stroke-width:2px
    style Supplemental Phase fill:#bbf,stroke:#333,stroke-width:2px
```

Data Submission

Once the data has been processed, evaluated, and validated, it is submitted to down-stream NHSN systems. A key feature of this process is the inclusion of **raw patient-level (aka: "line-level") data**.

This detailed data submission allows NHSN to perform further metrics and calculations at a granular level, providing deeper insights and more flexible analysis capabilities than aggregate reporting alone.

Facility/Tenant Configuration Requirements

Each facility or tenant in the NHSN reporting workflow must be configured with specific requirements to ensure reliable data flow and processing.

Data Source Configuration

The primary data source for NHSN reporting is a FHIR server. This requires:

- **Authentication**: Currently, we support **Epic** and **Cerner** authentication models, with plans to support more EHR vendors in the future.
- **Throttling**: Throttling configuration is critical to ensure that data acquisition does not negatively impact the facility's clinical systems during peak usage.

Census Acquisition Methods

The system supports multiple methods for determining the patient census, depending on the facility's technical capabilities:

- **FHIR Patient List**: Utilizing the Epic FHIR Patient List resource, if supported by the facility.
- **SFTP with CSV**: Securely transferring patient census data via CSV files over SFTP.
- **Future Support (Not Yet Available)**
  :
  - ADT via SFTP
  - Bulk Data
  - ADT via REST

Query Plan and Data Normalization

- **Query Plan**: A specific query plan must be defined for each facility to identify the exact FHIR resources and search parameters required for measure evaluation.
- **Normalization Operations**: Normalization processes are used to convert local codes and formats to standard codes (e.g., SNOMED-CT, LOINC, RXNORM) required for reporting.

Measure Configuration and Reporting Cadence

The reporting cadence and specific measures must be configured based on the facility's reporting requirements. Examples include:

- **NHSN Acute Care Monthly Initial Population**: Evaluating and submitting data on a monthly basis.
- **Daily Reporting**: Real-time or daily reporting for specific metrics to provide high-frequency insights.

Core Services Involved

The following core services are instrumental in the NHSN reporting workflow:

- DataAcquisitionService
- CensusService
- NormalizationService
- MeasureEvalService
- ValidationService
- SubmissionService

Telemetry

Type: Documentation

Telemetry is a key component of the system, providing observability into the performance, health, and behavior of the services. The system leverages a combination of tools and frameworks to collect, aggregate, and visualize trace, metric, and log data.

Key Components

- **Open Telemetry (OTEL)**:

  - Used by all services to report trace, metric, and log data.
  - Provides a standardized approach to telemetry data collection.

- **Loki and Prometheus**:

  - Loki: Used for collecting and managing logs.
  - Prometheus: Used for collecting and managing metrics.

- **OTEL Collector**:

  - Aggregates and collects trace, metrics, and logs data into a centralized location.
  - Acts as the intermediary, forwarding data to visualization and analysis tools.

- **Grafana**:

  - Visualizes the trace, metric, and log data.
  - Enables building and sharing dashboards for monitoring and analysis.
  - Dashboards for specific insights are actively being developed and can be exported/imported.

The components mentioned above are not part of this code base. They are open source components that are deployed as part of the applications architecture to support telemetry/observability.

Current Capabilities

- **Default Telemetry**:

  - Provides insights into CPU and RAM usage for all services.
  - Logs, metrics, and traces are collected and accessible for analysis.

- **Dashboards**:

    - Work is ongoing to create Grafana dashboards that provide actionable insights into system performance and health.

- **Correlation ID for Tracing**:

    - A newly generated correlationId is created when a request is initiated. This correlationId is passed along with subsequent requests, either via REST API headers or through Kafka message headers. This allows trace data to link related requests across services, providing end-to-end visibility into the flow of operations within the system.

- **Metrics**:

    - TODO: Document custom metrics that are already being collected.

Future Enhancements

- Exploration of additional metrics to further understand system health.
- Defining key indicators that provide deeper insights into the performance and reliability of the system.

Security

Type: Documentation

The Link system implements Role-Based Access Control (RBAC) through a claims-based authorization model. Currently, the system has a single "Admin" role configured which has access to all available permissions.

```
sequenceDiagram
    participant User as Application / User
    participant API as Link Admin API
    participant IDP as Identity Provider
    participant Account as Account Service

    User->>API: /api/login
    API->>IDP: Back channel request to /authorize endpoint
    API-->>User: Redirect to Identity Provider login
    User->>IDP: Authenticates
    IDP-->>API: Access Token
    API->>IDP: Back channel request to /userinfo endpoint
    IDP-->>API: Identity Token
    API->>Account: Back channel request to /account/api/user endpoint
    Account-->>API: Link User Claims
    API-->>User: Link Cookie *

    note over User,API: *Link Cookie is a session cookie with no expiration date set
    note over Account: Supported authentication schemes: OAuth, OpenIDConnect, JWT Bearer
```

Inter-service Authentication

Inter-service authentication is enforced using symmetric signing key on a token that is generated by the BFF and passed in requests to each of the micro-services.

Each service is provided access to the same signing key via the Secret Manager to sign tokens that are generated by each service.

There are two implementations of the Secret Manager:

- Azure Key Vault: Uses a centralized/cloud-based key vault to store the symmetric signing key.
- Local Secret Manager: Uses a per-service configuration to define the symmetric signing key used by each service.

The BFF has an option to `LinkTokenService:EnableTokenGenerationEndpoint` which allows users that are authenticated in the BFF to generate a token that can be passed directly to the micro services (for debugging purposes, for example). However, this should not (generally) be enabled in a production environment. In a production environment the BFF should be the only exposed service and the BFF would be responsible for proxying requests to each of the micro services.

When proxying a request from the BFF to micro services that were initiated by a user, the Yarp Proxy automatically generates a token to be passed in that proxy'd request to the micro service. You can control the life-span of the micro-service tokens generated by the BFF using the `LinkTokenService:TokenLifespan`

property. The LinkTokenService:LogToken property creates log entries when these tokens are generated by the BFF for debugging purposes.

Claims/Permissions

The following claims define what actions users can perform in the system:

Claim	Description
CanViewLogs	Allows viewing system audit and activity logs
CanViewNotifications	Allows viewing system notifications and alerts
CanViewTenantConfigurations	Allows viewing tenant configuration settings
CanEditTenantConfigurations	Allows modifying tenant configuration settings
CanAdministerAllTenants	Grants full administrative access across all tenants
CanViewResources	Allows viewing system resources
CanViewReports	Allows viewing generated reports
CanGenerateReports	Allows generating new reports
CanGenerateEvents	Allows generating system events
CanViewAccounts	Allows viewing user accounts
CanAdministerAccounts	Allows creating/modifying/deleting user accounts
IsLinkAdmin	Designates the user as a system administrator

Roles

Currently, only a single role is configured in the system:

Admin Role

- Has access to all claims/permissions listed above
- Full system administrative capabilities
- No tenant-level restrictions

Implementation

The RBAC system is implemented through:

- Claims defined in LinkSystemPermissions enum
- Authorization policies that map to individual claims
- Role and user entities that maintain claim assignments
- Claims-based authorization checks in the application

Future Considerations

Expanded Claims

The system is designed to support additional claims, particularly:

- UI-specific permissions for granular interface control
- Additional operational permissions as new features are added
- Workflow-specific permissions

Additional Roles

Plans for expanding role definitions include:

- Creating non-administrative roles with limited permissions
- Role hierarchies
- Custom role definitions per tenant

Tenant Restrictions

Future updates will include:

- Tenant-specific role definitions
- User-to-tenant mapping
- Tenant-scoped permissions
- Multi-tenant authorization policies

Retry Topics

Type: Documentation

The Link platform implements a robust retry mechanism for handling failed Kafka message processing across various services. This pattern uses dedicated retry topics, denoted with a "-Retry" suffix, to manage failed message processing attempts and ensure reliable message handling.

Implementation Components

Retry Topics

Retry topics are defined in the `KafkaTopic` enum and follow a consistent naming pattern:

- Original Topic: `{TopicName}`
- Retry Topic: `{TopicName}-Retry`

Example pairs:

- PatientEvent → PatientEvent-Retry
- ReportScheduled → ReportScheduled-Retry
- ResourceAcquired → ResourceAcquired-Retry

Core Components

RetryEntity

The `RetryEntity` class manages retry-related information including:

- Service name
- Facility ID
- Original topic
- Retry topic
- Retry count
- Next retry timestamp

RetryListener

A background service that:

- Consumes messages from retry topics
- Implements retry logic with exponential backoff
- Manages retry attempts and dead-letter handling
- Tracks retry counts and scheduling

RetryEntityFactory

Creates `RetryEntity` objects by:

- Processing message headers
- Calculating next retry attempts
- Managing retry topic naming
- Tracking retry metadata

Retry Flow

1. **Initial Processing**

- Service attempts to process message from primary topic
- If processing fails, message is sent to corresponding retry topic

2. **Retry Processing**

- RetryListener consumes from retry topic
- Validates retry count and timing
- Attempts reprocessing based on configured retry policy

3. **Retry Outcomes**

- Success: Message is processed and acknowledged
- Failure: Message is either:
- Scheduled for another retry if within retry limits
- Sent to dead-letter queue if retry limits exceeded

Configuration

Retry Settings

The retry settings are found in the `ConsumerSettings` property in each service's app/system configuration:

- DisableRetryConsumer: Disables the consumption of retry events
- RetryDuration: The duration for retry attempts. This is specified in [ISO 8601] format and is a list of retries. Each duration entry in the array represents the time to wait each time. If there are three entries, it will retry three times.

Header Management

Messages in retry topics include headers for:

- Original topic name

- Retry count
- Facility ID
- Service-specific metadata

Example Retry Topics

Common retry topics in the system include:

- AuditableEventOccurred-Retry
- DataAcquisitionRequested-Retry
- PatientEvent-Retry
- ReportScheduled-Retry
- ResourceAcquired-Retry
- SubmitReport-Retry

Each retry topic corresponds to a primary topic and follows the same processing pattern while maintaining service-specific handling requirements.

Performance Model

Type: Documentation

Overview

As Link scales from initial deployment to supporting tens of thousands of facilities, it is critical to have a clear performance model. This document outlines the throughput requirements at various milestones and identifies the key points in the reporting pipeline that must be monitored to ensure the system meets its performance goals.

Scale Milestones

The primary driver for performance is the number of facilities and the volume of patient data they produce. Based on an average of **12,000 patients per facility per month**, we can derive the required system throughput (patients per minute) for each milestone.

Assuming a month has approximately 44,000 minutes:

Milestone (Facilities)	Total Patients / Month	Required Throughput (Patients / Minute)
25	300,000	~6.8
100	1,200,000	~27.3
1,000	12,000,000	~272.7
10,000	120,000,000	~2,727.3
30,000	360,000,000	~8,181.8

Data Volume Scaling

Beyond patient counts, the total volume of FHIR resources processed by the **NormalizationService** and stored by **DataAccess** scales linearly with patient count.

Based on initial data analysis (averaging **708 resources per patient**):

- **100 Facilities**: ~850 Million resources / month (~19,300 resources / minute)
- **30,000 Facilities**: ~255 Billion resources / month (~5.8 Million resources / minute)

This highlights the necessity of a highly performant, distributed data lake (e.g., Iceberg) for the **DataAccess** domain.

Patient Payload Metrics (First Half)

In the First Half of the pipeline (Acquisition through Evaluation), the system processes individual patient data sets. The following metrics represent the volume of data (in serialized JSON format) associated with a single patient.

These metrics are based on an initial analysis of **10,765 patients** and approximately **7,622,175 resources**.

Metric	Size per Patient
Minimum	1.58 KB
Maximum	40.08 KB
Average	2.44 KB

Serialization Context

These metrics are derived using BSON.encode, which represents the **actual serialized bytes** of the data, rather than pretty-printed or indented JSON text. While Kafka events often include pretty-printing for human readability, this model focuses on the raw serialized size as it directly impacts storage requirements for **ReportService**'s database and intermediate Redis caching layers.

Encryption Impact

While encryption-at-rest is handled natively by the database (e.g., Cosmos DB/Mongo), the system must consider the impact of encrypting data in transient storage. Specifically, using **AES-128** to encrypt each resource before storing it in cache (e.g., Redis) adds a fixed overhead of approximately **32 bytes** per encryption operation.

Per-Resource in-transit encryption to support individual resource caching with an average of **708 resources per patient** adds **~22.7 KB** of metadata bloats every patient's footprint; increasing the average patient size from **2.44 KB** to **~25.14 KB** (a ~930% increase); not including processing timme for encrypting each resource.

In this model, where individual resources are small, per-resource encryption leads to significant "metadata bloat," where the encryption overhead (32 bytes) frequently exceeds the size of the resource itself.

Throughput Implications (First Half)

Based on the **Average (2.44 KB)** and **Maximum (40.08 KB)** payload sizes, we can project the data throughput requirements for the ingestion pipeline across different scales:

Milestone (Facilities)	Patients / Min	Avg Throughput / Min	Max Throughput / Min
100	~27.3	~66.6 KB	~1.1 MB
1,000	~272.7	~665.4 KB	~10.9 MB
10,000	~2,727.3	~6.7 MB	~109.3 MB
30,000	~8,181.8	~20.0 MB	~327.9 MB

These metrics inform the infrastructure requirements for transient data storage (e.g., Kafka message sizes, cache subscriptions) between the **DataAcquisitionService**, **NormalizationService**, and **MeasureEvalService**.

Reporting Pipeline Segmentation

To better analyze and scale the system, the reporting pipeline is divided into two primary segments:

First Half: Data Ingestion & Transformation

This segment covers the movement of data from the source EHR through the evaluation of measure logic. It is characterized by high-frequency, low-latency processing of individual patient payloads.

- **Services**: **DataAcquisitionService**, **DataAcquisitionWorkerService**, **NormalizationService**, **MeasureEvalService**
- **Workflow**: Acquisition -> Normalization -> Initial Evaluation -> (Optional) Supplemental Acquisition -> (Optional) Supplemental Normalization -> Supplemental Evaluation.

Second Half: Validation & Submission

This segment covers final quality checks and the actual submission to the reporting authority. While processing the same aggregate patient data volume as the first half, this phase typically handles larger bundles and performs high-latency operations like FHIR profile validation.

- **Services**: **ValidationService, SubmissionService**
- **Workflow**: Validation -> Submission.

Current Scaling & Capacity

As of version 0.5.0, the system is configured with the following resource allocations. These settings provide a baseline for the 25-100 facility milestones.

Service	Replicas	Primary Role
	10	High-compute CQL evaluation.
	10	High-frequency resource transformation.
	5	FHIR profile validation.
	3	Managing EHR query schedules.
	1	Executing EHR queries.
All Other Services	1	State management, UI, and low-traffic APIs.

Second Half Throughput (Validation)

Based on current performance monitoring (observing ReadyForValidation consumer lag), the **ValidationService** has the following capacity:

- **Avg Throughput**: ~1 patient every 2-3 seconds per replica.
- **Total Capacity (5 replicas)**: ~100-150 patients per minute.

This capacity is sufficient for the **100 Facilities** milestone (~27.3 patients/minute). However, to meet the **1,000 Facilities** milestone (~272.7 patients/minute), the **ValidationService** will need to scale to approximately 10-15 replicas, assuming linear scaling of the underlying FHIR validation engine.

Key Monitoring Points

Monitoring is divided between the two primary segments of the pipeline to better isolate performance bottlenecks.

First Half: Ingestion & Evaluation

1. Data Acquisition (EHR Interop)

- **Monitoring Point**: **DataAcquisitionService** and **DataAcquisitionWorkerService**.
- **Metric**: Time to retrieve patient data from external EHRs.
- **Constraints**: Limited to **8 threads per facility** to prevent EHR overload.
- **Milestone Impact**: At 30k facilities, even a small increase in EHR latency can lead to massive backlogs in the acquisition workers, as we are capped on concurrent requests per facility.

2. Normalization Throughput

- **Monitoring Point**: **NormalizationService**.
- **Metric**: Resources normalized per second.
- **Observation**: Tracked via **ResourceNormalized** processing times.
- **Milestone Impact**: Normalization is an atomic, high-frequency task. As patient volume grows, this service must scale horizontally to prevent being a bottleneck.

3. Measure Evaluation Latency

- **Monitoring Point**: **MeasureEvalService**.
- **Metric**: Time to evaluate CQL for a single patient (Initial and Supplemental).
- **Milestone Impact**: Evaluation is computationally intensive. Monitoring the `InitialEvaluation` and `SupplementalEvaluation` statuses in **ReportStatusUpdated** is critical.

Second Half: Validation & Submission

4. Validation Throughput

- **Monitoring Point**: **ValidationService**.
- **Metric**: Validation time per patient (targeting ~2-3 seconds per patient).
- **Observation**: Consumer lag on the `ReadyForValidation` topic.
- **Milestone Impact**: As reported in [Targeted Runtime Validation], full validation at the 30k scale is prohibitive. Monitoring the impact of "Unacceptable-only" validation rules on throughput is essential to maintain the 2-3 second/patient target.

5. Submission Success Rate

- **Monitoring Point**: **SubmissionService**.
- **Metric**: Submission success rate and latency.
- **Milestone Impact**: Final submission is the last mile. Monitoring for transient network errors or submission endpoint saturation is critical as the frequency of submissions increases.

Cross-Pipeline Health

6. Kafka Pipeline Health

- **Monitoring Point**: Kafka Broker and Consumer Groups.
- **Metric**: Consumer Lag (especially for ReportStatusUpdated and ResourceNormalized topics).
- **Milestone Impact**: As throughput increases to thousands of patients per minute, Kafka lag becomes the leading indicator of system saturation.

7. Report State Persistence

- **Monitoring Point**: **ReportService** and its underlying database.
- **Metric**: Database write latency for status updates.
- **Milestone Impact**: The ReportService must handle thousands of status updates per minute. Database indexing and write-ahead logging (WAL) performance will be key at the 10k+ facility scale.

EHR Acquisition Performance & Constraints

Real-world observations from our first facility deployments indicate that data acquisition from external EHRs is the most variable and constrained part of the pipeline. These metrics may vary from hospital to hospital depending on the underlying EHR infrastructure.

Resource Query Performance

Based on one week of data (25,125 data acquisition logs across 2,991 patients):

- **Avg Logs per Patient**: ~8.4
- **Avg Query Duration**: 5.32 seconds
- **Estimated Total Acquisition Time per Patient**: ~44.7 seconds

Query Performance by Resource Type

The following table summarizes the average latency and data volume for common FHIR resource queries. Some resources, like Observation, are significantly more expensive to retrieve and represent much larger payloads.

Resource Type	Avg Query Duration (Seconds)	Avg Resources Returned	Data Acquisition Logs
Observation	22.9	526	2,673
MedicationRequest	10.23	88	2,672
ServiceRequest	8.133	195	2,673
DiagnosticReport	2.50	43	2,673

Condition	2.14	10	2,673
Encounter	2.07	19	2,954
Device	1.64	3.4	470
Procedure	0.72	4.21	2,673
Coverage	0.44	4.47	2,673

Concurrency Constraints

To prevent overloading external EHR infrastructure and impacting hospital operations, concurrency is strictly limited:

- **Max Threads per Facility**: 8

Throughput Capacity (Facility Level)

Given an average patient acquisition time of ~44.7 seconds and a concurrency limit of 8 threads:

- **Max Patients / Minute / Facility**: ~10.7 patients

While this capacity is well above the average monthly requirement (~0.27 patients/minute), it limits the system's ability to "catch up" if backlogs occur or if reporting windows are short (e.g., daily submission cycles requiring all data to be pulled in a 2-hour window).

Observation & Tooling

We observe these metrics using the platform's standard telemetry stack as described in Telemetry.

- **Prometheus**: Collects custom metrics emitted by services (e.g., patient processing counts, phase durations).
- **Grafana**: Visualizes throughput vs. milestones. We should maintain a "Scale Dashboard" that compares current `Patients / Minute` against the targets defined in this model.
- **OTEL Tracing**: Provides end-to-end visibility using `correlationId` to identify where a specific patient's processing is stalled.
- **Report Performance Analysis**: Detailed granular tracking of pipeline phases is visualized in the **AdminUI** as proposed in Report Performance Analysis.

Related Entities

- Events: ReportStatusUpdated, ResourceNormalized
- Services: ReportService, DataAcquisitionService, MeasureEvalService, NormalizationService
- Proposals: Report State Machine Enhancements, Report Performance Analysis

Architecture References

Type: Documentation

Use this folder for system-wide architecture documentation such as auth flows, security posture, persistence strategy, telemetry, and scheduling.

Add new docs here when a change affects foundational architecture or shared platform concerns.

Dual Scheduler Architecture

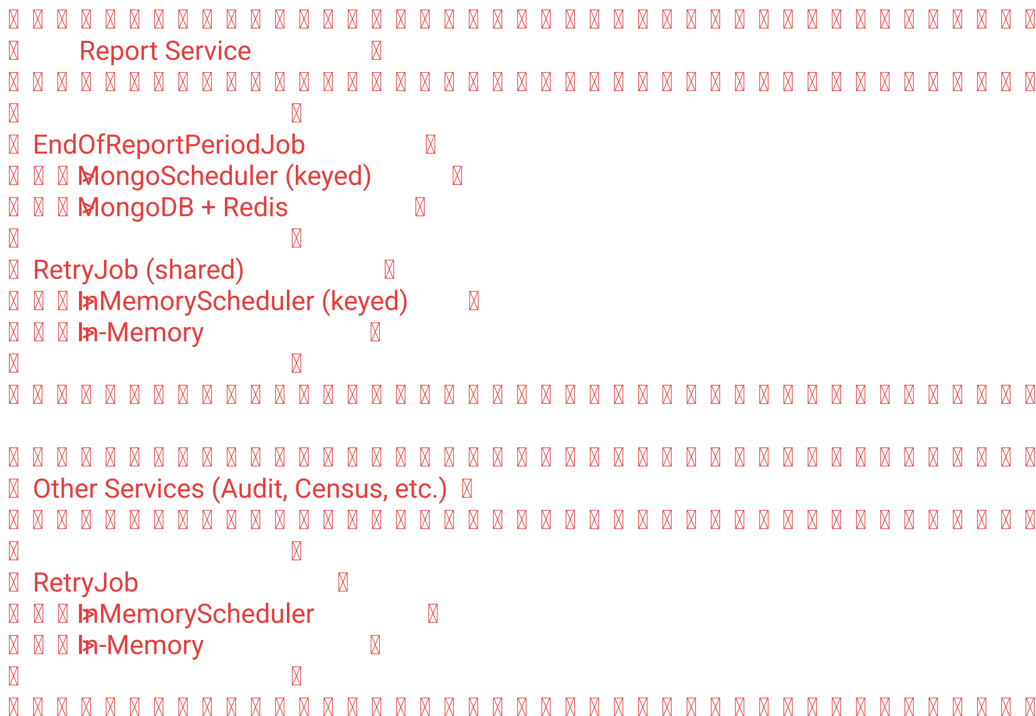
Type: Documentation

The Link Cloud solution implements a dual-scheduler architecture using Quartz.NET, enabling the Report service to use MongoDB-backed persistent job storage for clustering and fault tolerance, while maintaining lightweight in-memory scheduling for other services. This architecture provides proper isolation between different job types and enables horizontal scaling of the Report service.

Architecture Overview

The dual-scheduler system uses **keyed dependency injection** to register multiple `ISchedulerFactory` implementations:

- **MongoScheduler** - MongoDB-backed persistent storage (Report service only)
- **InMemoryScheduler** - RAM-based transient storage (all services)



MongoDB JobStore Configuration

Report Service Setup

The Report service uses Reddoxx.Quartz.MongoDbJobStore for persistent job storage with Redis-based distributed locking.

Service Registration

```
// Program.cs - Report service
builder.Services.AddQuartz(q =>
{
    q.UseJobFactory<JobFactory>();
    q.UseMicrosoftDependencyInjectionJobFactory();
});

// MongoDB scheduler for EndOfReportPeriodJob
builder.Services.AddKeyedSingleton<ISchedulerFactory>("MongoScheduler",
    (provider, key) =>
    {
        var logger = provider.GetRequiredService<ILogger<CustomMongoSchedulerFactory>>();
        return new CustomMongoSchedulerFactory(provider, logger);
    });

// In-memory scheduler for RetryJob
builder.Services.AddSingleton<InMemorySchedulerFactory>();
builder.Services.AddKeyedSingleton<ISchedulerFactory>("InMemoryScheduler",
    (provider, key) => provider.GetRequiredService<InMemorySchedulerFactory>());
```

CustomMongoSchedulerFactory

The CustomMongoSchedulerFactory creates and manages the MongoDB-backed scheduler:

```
public class CustomMongoSchedulerFactory : ISchedulerFactory
{
    public async Task<IScheduler> GetScheduler(CancellationToken cancellationToken = default)
    {
        // Create MongoDB job store
        var mongoJobStore = new MongoDbJobStore(loggerFactory, quartzFactory, _serviceProvider);

        mongoJobStore.CollectionPrefix = "reportjobs";
        mongoJobStore.Clustered = true;
        mongoJobStore.ClusterCheckinInterval = TimeSpan.FromMilliseconds(7500);
        mongoJobStore.ClusterCheckinMisfireThreshold = TimeSpan.FromMilliseconds(7500);

        // Initialize with distributed locking
        await mongoJobStore.Initialize(loadHelper, schedulerSignaler, cancellationToken);

        // Create scheduler
        DirectSchedulerFactory.Instance.CreateScheduler(
            schedulerName, schedulerInstanceId, threadPool, mongoJobStore);

        return await DirectSchedulerFactory.Instance.GetScheduler(schedulerName, cancellationToken);
    }
}
```

MongoDB Collections

The MongoDB JobStore creates the following collections:

Collection	Purpose
reportjobs.jobs	Job definitions with JobDataMap containing primitive values only

reportjobs.triggers	Trigger schedules, fire times, and trigger metadata
reportjobs.schedulers	Scheduler instance metadata for clustering
reportjobs.locks	Distributed locks managed by Redis for cluster coordination
reportjobs.calendars	Calendar-based exclusions for job scheduling
reportjobs.firedtriggers	Currently executing triggers across all instances
reportjobs.pausedtriggergroups	Paused trigger groups

Redis Distributed Locking

Redis provides distributed locking to prevent duplicate job execution across multiple Report service instances:

```
// Redis configuration
var redisConfiguration = new RedisConfiguration
{
    Hosts = new[] { new RedisHost { Host = redisHost, Port = redisPort } },
    Password = builder.Configuration["Redis:Password"],
    Database = 2
};
builder.Services.AddStackExchangeRedisExtensions<SystemTextJsonSerializer>(
    new[] { redisConfiguration });

// Register locking manager
builder.Services.AddSingleton<IQuartzJobStoreLockingManager,
    DistributedLocksQuartzLockingManager>();
```

In-Memory Scheduler Configuration

Services Using In-Memory Scheduling

The following services use in-memory scheduling for RetryJob:

- Audit
- Census
- Normalization
- QueryDispatch
- Submission

InMemorySchedulerFactory

The InMemorySchedulerFactory provides standard Quartz in-memory scheduling:

```
public class InMemorySchedulerFactory : ISchedulerFactory
{
    public async Task<IScheduler> GetScheduler(CancellationToken cancellationToken = default)
    {
        var properties = new NameValueCollection
        {
            { "quartz.scheduler.instanceName", "InMemoryScheduler" },
            { "quartz.scheduler.instanceId", $"{Environment.MachineName}-{DateTime.UtcNow.Ticks}" }
        };

        var schedulerFactory = new StdSchedulerFactory(properties);
        return await schedulerFactory.GetScheduler(cancellationToken);
    }
}
```

Service Registration Pattern

```
// Program.cs - All services except Report
builder.Services.AddSingleton<InMemorySchedulerFactory>();
builder.Services.AddKeyedSingleton<ISchedulerFactory>("InMemoryScheduler",
    (provider, key) => provider.GetRequiredService<InMemorySchedulerFactory>());
builder.Services.AddSingleton<ISchedulerFactory>(
    provider => provider.GetRequiredService<InMemorySchedulerFactory>());
```

Job Data Serialization

Problem: Complex Object Serialization

Previously, jobs stored entire model objects in the JobDataMap, causing BSON serialization failures:

```
// □ BEFORE - Causes MongoDB serialization errors
jobDataMap.Put("ReportScheduleModel", reportSchedule); // Full object
```

Solution: Store IDs, Fetch on Execution

Jobs now store only primitive identifiers and fetch full objects from the database when executing:

```
// □ AFTER - Store only IDs
public static IJobDetail CreateJob(ReportScheduleModel reportSchedule)
{
    JobDataMap jobDataMap = new JobDataMap();
    jobDataMap.Put("ReportScheduleId", reportSchedule.Id);
    jobDataMap.Put("FacilityId", reportSchedule.FacilityId);

    return JobBuilder
        .Create(typeof(EndOfReportPeriodJob))
        .StoreDurably(true)
        .RequestRecovery(true)
        .WithIdentity(reportSchedule.Id, "MeasureReportSubmissionGroup")
        .UsingJobData(jobDataMap)
        .Build();
```

```
}
```

Job Execution Pattern

Jobs retrieve fresh data from the database on each execution:

```
public async Task Execute(IJobExecutionContext context)
{
    // Get ID from JobDataMap
    string? scheduleId = context.JobDetail.JobDataMap.GetString("ReportScheduleId");

    if (string.IsNullOrEmpty(scheduleId))
    {
        // Fallback to trigger data map
        scheduleId = context.Trigger.JobDataMap?.GetString("ReportScheduleId");
    }

    // Fetch fresh data from database
    var schedule = await _database.ReportScheduledRepository.GetAsync(scheduleId);

    // Execute job logic with current data
    await ProcessReport(schedule);
}
```

Benefits

- **No Serialization Issues** - Only primitive types stored in MongoDB
- **Fresh Data** - Always works with current database state
- **No Stale Data** - Schedule updates reflected immediately
- **Smaller Footprint** - Minimal job data in MongoDB

Keyed Dependency Injection

Constructor Injection

Jobs and services use `FromKeyedServices` attribute to inject the correct scheduler:

```
// Report service - MongoDB scheduler
public class EndOfReportPeriodJob : IJob
{
    public EndOfReportPeriodJob(
        [FromKeyedServices("MongoScheduler")] ISchedulerFactory schedulerFactory,
        IDatabase database,
        ...)
    {
        _schedulerFactory = schedulerFactory;
    }
}

// Report listeners - MongoDB scheduler for job creation
public class ReportScheduledListener : BackgroundService
{
    public ReportScheduledListener(
```

```
        [FromKeyedServices("MongoScheduler")] ISchedulerFactory schedulerFactory,
        ...)
    {
        _schedulerFactory = schedulerFactory;
    }
}

// Retry services - In-memory scheduler
public class RetryScheduleService : BackgroundService
{
    public RetryScheduleService(
        [FromKeyedServices("InMemoryScheduler")] ISchedulerFactory schedulerFactory,
        ...)
    {
        _schedulerFactory = schedulerFactory;
    }
}
```

Configuration

Report Service Configuration

appsettings.json


```
{
  "MongoDB": {
    "ConnectionString": "mongodb://mongo:27017",
    "DatabaseName": "link-report"
  },
  "ConnectionStrings": {
    "Redis": "redis:6379"
  },
  "Redis": {
    "Password": "your-secure-password"
  },
  "Quartz": {
    "Scheduler": {
      "InstanceName": "ReportScheduler",
      "InstanceId": "AUTO"
    },
    "MongoDb": {
      "DatabaseName": "link-report",
      "CollectionPrefix": "reportjobs"
    }
  }
}
```


Docker Compose

```
report:
  environment:
    - MongoDB__ConnectionString=mongodb://mongo:27017
    - MongoDB__DatabaseName=link-report
    - ConnectionStrings__Redis=redis_cache:6379
    - Redis__Password=${REDIS_PASS}
  depends_on:
    - mongo
    - redis_cache
```

Required NuGet Packages

```
<ItemGroup>
  <PackageReference Include="Quartz" Version="3.*" />
  <PackageReference Include="Quartz.Serialization.Json" Version="3.15.0" />
  <PackageReference Include="Reddoxx.Quartz.MongoDbJobStore" Version="2.1.2" />
  <PackageReference Include="Reddoxx.Quartz.MongoDbJobStore.Redlock" Version="2.1.2" />
  <PackageReference Include="StackExchange.Redis.Extensions.Core" Version="11.0.0" />
  <PackageReference Include="StackExchange.Redis.Extensions.System.Text.Json" Version="11.0.0" />
  <PackageReference Include="StackExchange.Redis.Extensions.AspNetCore" Version="11.0.0" />
  <PackageReference Include="MongoDB.Driver" Version="3.2.0" />
</ItemGroup>
```

Other Services Configuration

Other services only require standard Quartz configuration:

```
{
  "Quartz": {
    "Scheduler": {
      "InstanceName": "InMemoryScheduler",
      "InstanceId": "AUTO"
    }
  }
}
```

Clustering & High Availability

Multiple Report Instances

The MongoDB JobStore with Redis locking enables safe operation of multiple Report service instances:

Instance Coordination

- **Cluster Check-in** - Every 7.5 seconds, each instance updates its status
- **Distributed Locks** - Redis-based locks prevent duplicate job execution
- **Instance Recovery** - Failed instances detected and their jobs reassigned
- **Load Distribution** - Jobs distributed across healthy instances

Monitoring Cluster Health

```
// Logs show cluster activity
[22:52:26 INF] ClusterManager: detected 1 failed or restarted instances.
[22:52:26 INF] ClusterManager: Scanning for instance "hostname-timestamp"'s failed in-progress jobs.
```

Failover Behavior

When a Report service instance fails:

1. Other instances detect the failure via cluster check-in timeout
2. Redis locks are automatically released
3. Jobs from the failed instance are marked for recovery
4. Healthy instances pick up the recovered jobs
5. Job execution continues without data loss

Job Scheduling Examples

Report Service - EndOfReportPeriodJob

```
public class MeasureReportScheduleService : BackgroundService
{
    public MeasureReportScheduleService(
        [FromKeyedServices("MongoScheduler")] ISchedulerFactory schedulerFactory,
        ...)
    {
        _schedulerFactory = schedulerFactory;
    }

    protected override async Task ExecuteAsync(CancellationToken cancellationToken)
    {
        var scheduler = await _schedulerFactory.GetScheduler(cancellationToken);
        scheduler.JobFactory = _jobFactory;

        // Find all reports that haven't completed
        var reportSchedules = await _database.ReportScheduledRepository
            .FindAsync(s => !s.EndOfReportPeriodJobHasRun &&
                        s.Frequency != Frequency.Adhoc,
                    cancellationToken);

        foreach (var reportSchedule in reportSchedules)
        {
            await CreateJobAndTrigger(reportSchedule, scheduler);
        }

        await scheduler.Start(cancellationToken);
    }

    public static async Task CreateJobAndTrigger(
        ReportScheduleModel reportSchedule,
        IScheduler scheduler)
    {
        var job = CreateJob(reportSchedule);
        var trigger = CreateTrigger(reportSchedule, job.Key);

        var exists = await scheduler.CheckExists(job.Key);
        if (!exists)
        {
            await scheduler.ScheduleJob(job, trigger);
        }
    }
}
```

All Services - RetryJob

```
public class RetryScheduleService : BackgroundService
{
    public RetryScheduleService(
        [FromKeyedServices("InMemoryScheduler")] ISchedulerFactory schedulerFactory,
        ...)
    {
        _schedulerFactory = schedulerFactory;
    }

    protected override async Task ExecuteAsync(CancellationToken cancellationToken)
    {
        var scheduler = await _schedulerFactory.GetScheduler(cancellationToken);
        scheduler.JobFactory = _jobFactory;

        var retries = await _retryRepository.GetAllAsync(cancellationToken);

        foreach (var retry in retries)
        {
            await CreateJobAndTrigger(retry, scheduler);
        }

        await scheduler.Start(cancellationToken);
    }
}
```

Migration Guide

Deploying the Dual Scheduler

Prerequisites

1. **MongoDB** - Accessible from Report service instances
2. **Redis** - Accessible from Report service instances
3. **Configuration** - Update appsettings.json with connection strings

Deployment Steps

1. **Update Configuration**

```
# Set MongoDB connection string
export MongoDB__ConnectionString="mongodb://mongo:27017"
export MongoDB__DatabaseName="link-report"

# Set Redis connection
export ConnectionStrings__Redis="redis:6379"
export Redis__Password="secure-password"
```

2. **Deploy Report Service**

```
# MongoDB collections are auto-created on first startup
docker-compose up -d report
```

3. **Verify Deployment**

```
# Check MongoDB collections exist
mongo link-report --eval "db.getCollectionNames()"
# Should show: reportjobs.jobs, reportjobs.triggers, etc.
```

```
# Check scheduler logs
docker logs link-report | grep "Scheduler created successfully"
```

Important Notes

- **Existing Jobs** - In-memory jobs are NOT automatically migrated to MongoDB
- **Rescheduling** - Existing report schedules will be picked up on first startup
- **Storage** - Monitor MongoDB storage growth, implement cleanup policies if needed
- **Clustering** - Multiple Report instances can be deployed immediately

Rollback Plan

If issues occur, rollback to in-memory scheduling:

```
// Temporarily revert to in-memory for Report service
builder.Services.AddSingleton<InMemorySchedulerFactory>();
builder.Services.AddKeyedSingleton<ISchedulerFactory>("MongoScheduler",
    (provider, key) => provider.GetRequiredService<InMemorySchedulerFactory>());
```

Monitoring & Troubleshooting

Key Metrics to Monitor

MongoDB

- Collection sizes (reportjobs.*)
- Document counts in jobs and triggers collections
- Query performance for job retrieval
- Storage usage trends

Redis

- Lock acquisition/release rates
- Lock contention events
- Connection pool usage
- Memory usage

Scheduler Health

- Job execution success/failure rates
- Misfire counts
- Cluster instance count
- Instance check-in frequency

Common Issues

Issue: Jobs Not Persisting

Symptoms: Jobs execute but don't appear in MongoDB

Solutions:

- Verify MongoDB connection string is correct
- Check MongoDB user has write permissions
- Confirm `CollectionPrefix` setting matches
- Review logs for BSON serialization errors

Issue: Duplicate Job Execution

Symptoms: Same job runs multiple times across instances

Solutions:

- Verify Redis is accessible from all instances
- Check Redis distributed locking is enabled
- Confirm `Clustered = true` in job store config
- Review Redis logs for connection issues

Issue: Cluster Instance Not Detected

Symptoms: Warning about failed instances repeatedly

Solutions:

- Clean up stale scheduler records in MongoDB

```
db.reportjobs.schedulers.deleteMany({})
db.reportjobs.locks.deleteMany({})
```

- Restart all Report service instances
- Verify cluster check-in interval settings

Issue: BSON Serialization Errors

Symptoms: `BsonSerializationException` for custom types

Solutions:

- Ensure only primitive types in JobDataMap
- Register custom types with BSON serializer if needed

```
MongoDB.Bson.Serialization.BsonClassMap.RegisterClassMap<YourType>(cm =>
{
    cm.AutoMap();
    cm.SetIgnoreExtraElements(true);
});
```

Security Considerations

MongoDB Access

- Use MongoDB authentication in production
- Limit MongoDB user permissions to Report database only
- Enable MongoDB SSL/TLS connections
- Use network isolation (VPC/VNET) when possible

Redis Access

- Always use Redis password authentication
- Consider Redis ACLs for fine-grained access control
- Enable Redis SSL/TLS in production
- Restrict Redis network access to Report service instances

Job Data Security

- Never store sensitive data in JobDataMap
- Store only identifiers, fetch sensitive data on execution
- Consider encrypting sensitive database fields
- Audit job execution logs for compliance

The dual-scheduler architecture provides production-ready job scheduling with persistence, clustering, and fault tolerance for the Report service, while maintaining lightweight in-memory scheduling for retry operations across all services.

Data Persistence

Type: Documentation

This page provides a comprehensive overview of the data persistence strategy across the Link platform. Persistence is categorized into three main types, each serving specific architectural needs.

Persistence Strategy Overview

The Link platform utilizes a polyglot persistence approach to optimize for different data characteristics:

```
graph TD
    %% Databases
    SQL[(SQL Server)]
    Mongo[(MongoDB)]
    ABS[(Azure Blob Storage)]

    %% Services
    AccountService[[Account Service]]
    AuditService[[Audit Service]]
    CensusService[[Census Service]]
    DAService[[Data Acquisition Service]]
    NormService[[Normalization Service]]
    NotifService[[Notification Service]]
    TenantService[[Tenant Service]]
    MEService[[Measure Evaluation Service]]
    ReportService[[Report Service]]
    SubService[[Submission Service]]
    ValService[[Validation Service]]

    %% Connections to SQL
    AccountService --- SQL
    AuditService --- SQL
    CensusService --- SQL
    DAService --- SQL
    NormService --- SQL
    NotifService --- SQL
    TenantService --- SQL
    ValService --- SQL
    SubService --- SQL

    %% Connections to MongoDB
    MEService --- Mongo
    ReportService --- Mongo

    %% Connections to ABS
    MEService --- ABS
    SubService --- ABS
    ValService --- ABS

    classDef database fill:#f9f,stroke:#333,stroke-width:2px;
    class SQL,Mongo,ABS database;
```

- **SQL Server**: The primary store for **configuration and metadata**. Most services use SQL Server to maintain state, tenant settings, and operational logs.
- **MongoDB (Azure Cosmos DB)**: Deployed as Azure Cosmos for MongoDB, this layer is used for **clinical data and reporting artifacts**. It handles the high-volume, semi-structured nature of FHIR resources and evaluation results.
- **Azure Blob Storage**

: Used for
large file persistence
(e.g., patient bundles, submission NDJSON files).
- *Platform Agnosticism*: Blob storage access is abstracted into interface patterns (e.g., `IBlobStorageService`) to ensure the platform remains agnostic of the underlying cloud provider.
- *Usage*: Used for internal submissions within the reporting pipeline and external submissions to public health agencies.

Service-Specific Persistence

Detailed persistence schemas for each service are available directly on their respective documentation pages under the "Database Schema" section. These schemas are maintained as JSON Schema files (`db.schema.json`) within each service's directory.

SQL Server (Relational)

The following services utilize SQL Server for structured configuration and metadata:

- **AccountService**
- **AuditService**
- **CensusService**
- **DataAcquisitionService**
- **NormalizationService**
- **NotificationService**
- **SubmissionService**
- **TenantService**
- **ValidationService**

MongoDB (Document-based)

Used for high-volume clinical and reporting data:

- **MeasureEvalService**
- **ReportService**

Azure Blob Storage (Object Storage)

Used for large artifacts and submission bundles:

- **MeasureEvalService** (evaluated reports)
- **SubmissionService** (internal and external bundles)
- **ValidationService** (validation results)

Reporting Pipeline Data Flow

The following diagram illustrates the flow of data through the reporting pipeline and the specific points where it is persisted to the various database layers:

```
graph TD
    subgraph Pipeline ["Reporting Pipeline Flow"]
        DAService[[Data Acquisition Service]] -->|Acquires| NormService[[Normalization Service]]
        NormService -->|Normalizes| MEService[[Measure Evaluation Service]]
        MEService -->|Evaluates| ReportService[[Report Service]]
        ReportService -->|Validates| ValService[[Validation Service]]
        ValService -->|Submits| SubService[[Submission Service]]
    end

    %% Persistence
    Mongo[(MongoDB)]
    IntABS[(Internal ABS)]
    ExtABS[(External ABS)]

    %% Connections
    MEService -.->|Persist Normalized Data| Mongo
    ReportService -.->|Persist Evaluated Reports| IntABS
    SubService -.->|Persist Final Submissions| ExtABS

    classDef service fill:#fff,stroke:#333,stroke-width:2px;
    classDef storage fill:#f9f,stroke:#333,stroke-width:2px;
    class DAService,NormService,MEService,ReportService,ValService,SubService service;
    class Mongo,IntABS,ExtABS storage;
```

Data Persistence Lifecycle

1. **Acquisition & Normalization**: Data is retrieved from EHRs and normalized to ensure FHIR compliance.
2. **Measure Evaluation**: The normalized data is passed to the **MeasureEvalService**, where it is **persisted in MongoDB** for evaluation processing.
3. **Evaluation & Reporting**: After the CQL engine evaluates the data, the resulting MeasureReports are processed by the **ReportService** and **persisted in Internal Blob Storage**.
4. **Validation & Submission**: The evaluation results are validated against FHIR profiles and quality measures. Once validated, the **SubmissionService** packages the content and **persists the final bundles in External Blob Storage** for delivery to public health agencies.

Use the GH-link-cloud tool to identify the persistence points across all of the services and to create an AVRO schema for each service that shows each of the tables/collections and their columns. Then create a "Data Persistence" custom page that describes the persistence layers (including blob storage in Measure Eval and Submission) across the platform as a whole. I'd like each service to have a good level of detail (the AVRO schema) for the database of each service, and a good summary of data persistence across the platform in the custom doc. There are three types of persistence: * SQL Server (most services use this at least for configuration) * MongoDB (deployed as Azure Cosmos for MongoDB) used by MeasurEval and Report services * Azure Blob Storage (which needs to be abstracted out into interface patterns so that we can stay platform agnostic) used for internal submissions and eventually external submissions after patient data has completed moving through the reporting pipeline   Instead of putting json blocks in the data-persistence.mdx file, put each .avsc avro file in its associated service directory (i.e. domains/Compliance/AccountService/db.avsc) and add `<Schema file="db.avsc" title="Database Schema" />` to each of the service's index.mdx files Instead of <Schema> use <SchemaViewer> on all pages. Narrative description of the service should show before the schema viewer. Rename all of the .avsc files to .avro   The AVRO schemas are displaying in the viewer, but they're only showing the first

level of properties. I can't expand each property to see the sub-properties. I don't want to rely on custom changes to files in `.eventcatalog-core` to view the schema successfully. Switch from using AVRO for the schema definitions to regular JSON schema. Rename the .avro files to db.schema.json and change the content to JSON schema format.

Auth Flow

Type: Documentation

The Link Admin BFF (Backend for Frontend) implements a secure authentication flow using OAuth 2.0 with cookie-based session management. This provides a robust security model where the BFF acts as the OAuth client and manages user sessions through secure HTTP-only cookies.

Authentication Flow

1. **Initial Login Request**

- The UI redirects users to `/api/login` on the BFF
- The BFF initiates the authorization by redirecting to the configured Identity Provider (IdP) or Authorization Server
- The OAuth provider authenticates the user and returns an authorization code

2. **Token Exchange & Session Creation**

- The BFF exchanges the authorization code for access/refresh tokens
- The BFF creates a session and issues a secure HTTP-only cookie named "link_cookie"
- The cookie contains session information but not the actual OAuth tokens
- OAuth tokens are securely stored server-side and managed by the BFF

3. **Subsequent Requests**

- The UI includes the session cookie automatically with each request
- The BFF validates the cookie and retrieves the associated session
- The BFF uses the OAuth tokens to make authenticated requests to backend services
- Token refresh is handled transparently by the BFF when needed
- The BFF generates a short lived JWT token and adds it to the Authorization header when making requests to backend services

Cookie Security

- Cookies are HTTP-only to prevent XSS attacks
- Secure flag ensures cookies only sent over HTTPS
- Anti-forgery protection enabled
- Session cookies with server-side storage

Token Security

- OAuth tokens never exposed to the frontend
- Tokens stored securely server-side
- Token refresh handled by BFF
- Short-lived access tokens with automatic refresh

CORS Protection

- Strict CORS policy configured
- Credentials mode enabled for cookie transmission
- Only trusted origins allowed

PKCE (Proof Key for Code Exchange)

PKCE is implemented to secure the OAuth authorization code flow and prevent authorization code interception attacks:

Code Verifier Generation

- A cryptographically random code verifier is generated for each authentication request
- The verifier is stored securely server-side in the BFF
- Length and character requirements follow OAuth PKCE specifications

Code Challenge

- The code challenge is derived from the verifier using SHA-256
- Challenge is included in the initial authorization request
- The challenge method is set to "S256" (SHA-256)

Code Exchange

- Original code verifier is included when exchanging the authorization code
- OAuth provider validates the code challenge/verifier pair
- Prevents malicious actors from using intercepted auth codes

State Parameter Protection

The state parameter provides CSRF protection during the OAuth flow:

State Generation

- Cryptographically secure random state value generated per request
- State is bound to the user's session
- Stored server-side in the BFF

State Validation

- State parameter included in authorization request
- Returned state must match the original stored value
- Requests with invalid/missing state are rejected
- Protects against CSRF and replay attacks

Scope Restrictions

OAuth scopes are strictly controlled to limit access:

Required Scopes

openid profile email

Scope Validation

- Only pre-configured scopes are allowed
- Scope requests are validated against allowed list
- Additional scopes require explicit configuration
- Prevents scope elevation attacks

Scope Usage

- Scopes map to specific API permissions
- Access tokens only receive approved scopes
- Resource access is restricted by granted scopes

Configuration

The BFF's OAuth and cookie settings are configured in `appsettings.json`:

```
"Authentication": {
  "DefaultScheme": "link_cookie",
  "DefaultChallengeScheme": "link_oauth2",
  "Schemas": {
    "Cookie": {
      "HttpOnly": true,
      "Domain": "",
      "Path": "/"
    },
    "Oauth2": {
      "Enabled": true,
      "ClientId": "",
      "ClientSecret": "",
      "Endpoints": {
        "Authorization": "",
        "Token": "",
        "UserInformation": ""
      },
      "CallbackPath": "/api/signin-oauth2"
    }
  }
}
```

UI Integration

The UI application interacts with the BFF's authentication system through:

Login

```
// Redirect to BFF login endpoint
window.location.href = '/api/login';
```

Session Status

```
// Check if user is authenticated
fetch('/api/user', {
credentials: 'include' // Important for cookie transmission
});
```

Logout

```
// Redirect to BFF logout endpoint
   window.location.href = '/api/logout';
```

The UI never directly handles OAuth tokens or credentials - all authentication is delegated to and managed by the BFF or a designated identity provider such as Azure or Keycloak.

Deployment & Usage

Type: Documentation

Docs in this folder focus on getting Link running in an environment and using it day to day.

- Config: Service configuration references (env vars, runtime settings, integration config).
- Usage: Operator and tenant workflows, runbooks, and UI usage guidance.

Tenant Management

Type: Documentation

In Link Cloud, the terms "tenant" and "facility" are used interchangeably to refer to healthcare organizations configured within the system. Each tenant/facility is identified by a unique `facilityId` that must be consistent across all services to ensure proper functionality.

Tenant Service

The Tenant service acts as the primary entry point for configuring organizations within Link Cloud. It maintains core facility configurations and generates events for scheduled measure reporting periods.

Core Tenant Configuration Elements

- Facility ID (unique identifier used across all services; two modes: NHSN ORG ID (5 digits) or Custom ID (alphanumeric plus hyphens))
- Facility Name
- Creation and Modification dates
- Scheduled Tasks
- Monthly Reporting Plans
  - Report Types
  - Reporting Periods
  - Submission Schedules

Cross-Service Tenant Configuration

A fully functional tenant requires configuration across multiple services. Each service maintains its own tenant-specific configurations using the `facilityId` as the linking identifier.

Service-Specific Configurations

Tenant Service

- Basic facility information
- Reporting schedules
- Monthly reporting plans
- Multi-measure reporting configurations
- NHSN ORG ID enforcement flag
- Soft delete support (`isDeleted` flag)

Data Acquisition Service

- Data source configurations

- Query parameters
- Resource type settings
- Data collection schedules

Measure Evaluation Service

- Measure specifications
- Evaluation schedules
- Performance settings
- Output configurations

Report Generation Service

- Report templates
- Output format preferences
- Delivery settings
- Generation schedules

Notification Service

- Email notification settings
- Recipient lists
- Alert preferences
- Communication channels

Census Services

- Facility identification
- Facility ID linking
- Tenant API endpoint configuration
- Base service URLs
- Patient Census Management
- FHIR List query frequency settings
- Admission tracking parameters
- Discharge monitoring configurations
- Patient information retention policies

Query Dispatch Service

- Facility Configuration
- Facility ID mapping
- FHIR endpoint connections
- Resource query parameters
- Query Timing Management
- Discharge lag period settings
- Resource query scheduling
- Data settlement wait times
- Query retry parameters
- Resource Query Settings
- FHIR resource type configurations
- Query batch sizes

- Performance optimization parameters
- Resource filtering rules

Important Notes

1. Creating a tenant configuration in the Tenant service alone does not create a fully functional tenant. Additional configuration in other services is required.
2. The facilityId must be consistent across all services to ensure proper system integration and functionality.
3. Each service captures and manages different aspects of a tenant's configuration, working together to provide complete functionality.

Best Practices

Configuration Management

- Ensure all required services are properly configured for each tenant
- Maintain consistent facilityId usage across services

Link Product Overview

Type: Documentation

Link Product Whitepaper: Modernizing Healthcare Data Reporting

Executive Summary

Link (also known as NHSNLink) is a cloud-native, event-driven platform designed to automate and streamline the complex process of healthcare data reporting. By leveraging FHIR® (Fast Healthcare Interoperability Resources) and a modular microservices architecture, Link enables healthcare organizations to move from burdensome manual reporting to automated, scalable, and highly accurate data submission pipelines.

The Problem: The High Cost of Manual Reporting

Healthcare facilities today face significant challenges in meeting reporting requirements from public health agencies like the CDC's National Healthcare Safety Network (NHSN):

- **Manual Data Collection**: Infection preventionists often spend hours manually extracting data from various Electronic Health Records (EHRs).
- **Data Fragmentation**: Clinical data is siloed and often exists in incompatible formats, making it difficult to aggregate and evaluate.
- **Complex Compliance**: Reporting requirements and clinical measure definitions (e.g., CQL) are constantly evolving, requiring frequent system updates.
- **Lack of Traceability**: Traditional reporting methods often lack the audit trails needed to ensure data integrity and reproducibility.
- **Scalability Issues**: Existing on-premise solutions often struggle with the high volume of clinical data generated in modern healthcare environments.

The Solution: Link

Link addresses these pain points by providing an automated, scalable, and extensible platform for the entire data reporting lifecycle.

Core Capabilities

- **Automated Data Acquisition**: Link supports both PUSH and PULL models for acquiring clinical data from EHRs, ensuring that the latest patient information is always available for evaluation.
- **FHIR-Native Processing**: All data within Link is normalized and processed as FHIR resources, ensuring interoperability and compliance with modern healthcare standards.
- **Modular Reporting Pipelines**: Reporting workflows are orchestrated through configurable services, allowing Link to adapt to various reporting requirements and use cases.
- **Advanced Measure Evaluation**: Link integrates a CQL (Clinical Quality Language) engine to evaluate complex clinical quality measures against normalized data.

- **Multi-Tenancy and Isolation**: Built from the ground up for the cloud, Link supports multi-tenant deployments, ensuring secure data isolation for different healthcare organizations.
- **Automated Submission**: Once validated, reports are automatically packaged and submitted to public health agencies, reducing manual effort and minimizing errors.

Technical Architecture

Link's architecture is built on modern principles to ensure performance and reliability:

- **Event-Driven Design**: The system uses an event-driven architecture to decouple services and ensure high availability and scalability.
- **Polyglot Persistence**: Link optimizes data storage by using SQL Server for metadata, MongoDB (Azure Cosmos DB) for clinical data, and Azure Blob Storage for large artifacts.
- **Audit and Compliance**: Built-in auditing and validation services ensure that every step of the reporting process is documented and compliant with defined policies.

Key Benefits

- **Efficiency**: Significantly reduces the time and effort required for clinical reporting.
- **Accuracy**: Minimizes human error through automated data extraction and evaluation.
- **Scalability**: Easily handles large volumes of data across multiple facilities.
- **Flexibility**: Quickly adapts to new reporting requirements through its modular design.
- **Compliance**: Ensures data is handled securely and reported accurately according to the latest standards.

Conclusion

Link is more than just a reporting tool; it is a comprehensive platform for healthcare data interoperability and analytics. By automating the data reporting lifecycle, Link empowers healthcare organizations to focus on what matters most: improving patient care.

Navigation

- **DataAcquisitionService**: Learn more about how Link acquires data.
- **ReportService**: Explore the reporting orchestration.
- **MeasureEvalService**: Understand how clinical measures are evaluated.
- **TenantService**: Manage tenants and facilities.
- [Deployment Configuration]: View technical configuration guides.
- [User Guides]: Operator and tenant workflows.

Configuring Java Services

Type: Documentation

Spring Boot uses two main YAML configuration files:

- bootstrap.yml
  - Loaded before application.yml during the startup process.
  - Used for early-stage configuration that is required before the main application context is loaded.

- application.yml
  - Loaded after bootstrap.yml, within the main application context.

Some properties may want to be put in environment-specific configuration files with the naming convention bootstrap-ENV.yml and application-ENV.yml. For example, there may be bootstrap-dev.yml and bootstrap-prod.yml. At deployment-time, you can specify an environment variable SPRING_PROFILES_ACTIVE to indicate whether to load dev or prod on top of the default configurations.

To convert a property name in the canonical-form to an environment variable name you can follow these rules:

- Replace dots (.) with underscores (_).
- Remove any dashes (-).
- Convert to uppercase.

For example, the configuration property spring.main.log-startup-info would be an environment variable named SPRING_MAIN_LOGSTARTUPINFO.

Environment variables can also be used when binding to object lists. To bind to a List, the element number should be surrounded with underscores in the variable name.

For example, the configuration property my.service[0].other would use an environment variable named MY_SERVICE_0_OTHER.

Examples of environment variable naming conventions:

YAML / JSON Key	Converted Environment Variable
server.port	SERVER_PORT
spring.datasource.url	SPRING_DATASOURCE_URL
spring.cloud.azure.appconfiguration.stores[0].selects[0].label-	SPRING_CLOUD_AZURE_APPCONFIGURATION_STORES_0_S

filter	
my.custom.settings[2].api-key	MY_CUSTOM_SETTINGS_2_APIKEY
config.services[1].endpoints.internal-url	CONFIG_SERVICES_1_ENDPOINTS_INTERNALURL

Note: Official guidance is to *remove* dashes (-) entirely from the environment variable. For example: label-filter becomes LABELFILTER. However, both LABEL-FILTER and LABELFILTER work interchangeably.

Overriding Default Configs

Any of the properties for serivce configuration can be provided either via environment variables, through a custom application.yml file, or via properties set in java using -D<propertyName>=<value> passed as an argument to the JVM during startup.

Azure App Config and Key Vault

Default key-filter and label-filter properties are specified for each service, so that at deployment time only the connection to the Azure App Config or Key Vault services needs to be configured/specified in environment variables.

Property Name	Description	Type/ Value
spring.cloud.azure.appconfiguration.enabled	Enable Azure App Configuration	true or false
spring.cloud.azure.appconfiguration.stores[0].connection-string	Connection string to Azure App Config instance (if not using managed identity).	<string>
spring.cloud.azure.appconfiguration.stores[0].endpoint	Endpoint to use for App Config when managed identity should be specified via AZURE_CLIENT_ID	<string>
spring.cloud.azure.appconfiguration.stores[0].selects[0].label-filter	Label to use for configuration	",Validation"
spring.cloud.azure.appconfiguration.stores[0].selects[0].key-filter	Key to use for configuration	"/"

Authentication

Java Azure libraries have difficult using different authentication mechanisms between App Config (AAC) and Key Vault (AKV). If you specify AZURE_CLIENT_ID, it will attempt to use managed identity for *both* AAC and AKV.

If using managed identity authentication for one, it is suggested to use managed identity for both; *not* a connectionString with a token/secret embedded in it for AAC and MI for AKV.

Specifying all three AZURE_CLIENT_ID, AZURE_CLIENT_SECRET and AZURE_TENANT_ID is only necessary when using a service principal for authentication. Only AZURE_CLIENT_ID is necessary to authenticate using managed identity.

If using a service principal for authentication, the AZURE_TENANT_ID is *not* the same as the subscription ID.

Blob Storage

Property Name	Description	Required	Default Value	Secret?
internal-blob-storage.connection-string	Connection string for internal ABS	Yes (if using ABS)	None	Yes
internal-blob-storage.blob-container-name	Blob container name for internal ABS	Yes (if using ABS)	None	No

Telemetry

Property Name	Description	Type/Value
telemetry.exporterEndpoint	Endpoint that can be connected to by scrapers for metric data	"http://localhost:55690"
loki.enabled	Enable Loki for logging	true or false
loki.url	URL for Loki	"http://localhost:3100"
loki.app	Application name for Loki	"link-dev"

Swagger

Property Name	Description	Type/Value
springdoc	Configuration for Swagger and Swagger UI	See Springdoc documentation for details
springdoc.api-docs.enabled	Enable Swagger specification generation	true or false (default)
springdoc.swagger-ui.enabled	Enable Swagger UI	true or false (default)

Databases

Mongo DB

Property Name	Description	Type/Value	Secret?
spring.data.mongodb.host	Host address for the Mongo database	"localhost"	No
spring.data.mongodb.port	Port for the Mongo database	27017	No
spring.data.mongodb.database	Database name for the Mongo database	"measureeval"	No
spring.data.mongodb.username	Username for the Mongo database	<string>	No
spring.data.mongodb.password	Password for the Mongo database	<string>	Yes

SQL Server

Property Name	Description	Type/Value	Secret?
spring.datasource.url	URL for the SQL Server database	<string> prefixed with "jdbc:sqlserver://"	No

spring.datasource.username	Username for the SQL Server database	<string>	No
spring.datasource.password	Password for the SQL Server database	<string>	Yes
spring.jpa.hibernate.ddl-auto	DDL auto setting for JPA/ Hibernate	"none" (default) or "update"	No
spring.jpa.properties.show_sql	Show SQL statements in logs	true (default) or false	No
spring.jpa.properties.dialect	SQL dialect for the database	"org.hibernate.dialect.SQLServerDialect" (default)	No

Auto Update/Migrate DBs

Property Name	Description	Type/Value	Secret?
spring.jpa.hibernate.ddl-auto	Indicates whether how to update the schema in hibernate databases	create | create-drop | update | validate | none	No

Kafka

Property Name	Description	Type/Value	Secret?
spring.kafka.bootstrap-servers	Kafka bootstrap servers	"localhost:9092"	No
spring.kafka.consumer.group-id	Kafka consumer group ID	"measureeval"	No
spring.kafka.producer.client-id	Kafka producer client ID	"measureeval"	No
spring.kafka.retry.maxAttempts	Maximum number of times consumption of an event should be retried	3	No
spring.kafka.retry.retry-backoff-	Time in milliseconds to wait before retrying a	3000	No

ms	failed event

Service Authentication

Property Name	Description	Type/Value	Secret?
secret-management.key-vault-uri	URI for the Azure Key Vault	<string>	Yes
authentication.adminEmail	Email address representing the Link administrator account	<string>	No
authentication.anonymous	Whether the service should allow anonmyous users access to the services. This should onyl be enabled for DEV environments.	true or false (default)	No
authentication.authority	Authority for the service to authenticate against.	"http://localhost:7004"	No
authentication.signingKey	Signing key for generating/verifying JWTs	<string>	Yes

HAPI FHIR Server Configuration

The HAPI module supports rate-limiting configuration to control the number of requests allowed within a specified time period. When rate-limiting is enabled, clients that exceed the configured limits will receive HTTP 429 (Too Many Requests) responses.

Property Name	Description	Type/Value	Secret?
hapi.fhir.rate-limiting.count	Maximum number of requests allowed per client within the specified duration. Set to -1 to disable rate-limiting entirely.	Integer (default: -1)	No
hapi.fhir.rate-limiting.duration	Duration for the rate-limiting window. Uses ISO-8601 duration format.	Duration (default: PT1M)	No

Environment Variable Examples:

- `HAPI_FHIR_RATELIMITING_COUNT=100` - Allow 100 requests per time window

- HAPI_FHIR_RATELIMITING_DURATION=PT5M - Set time window to 5 minutes

Note: Rate-limiting is applied per client IP address. When count is set to -1 (default), no rate-limiting is applied.

Service Information

The service-information configuration allows exposing build, version and deployment information through the /api/XXX/info (i.e. /api/validation/info) endpoint of each service. This information is automatically populated from assembly metadata and git information during build time.

Property	Description	Required	Default Value	Secret?
service-information.service-name	Display name of the service	No	Varies service by service	No
service-information.version	Version of the service	No	Version from java/maven's BuildProperties	No
service-information.product-version	Version of the product as a whole	No	N/A	No
service-information.build	Build number from CI/CD pipeline	No	"dev" when using Docker Compose	No
service-information.commit	Git commit hash of deployed version	No	None	No

Deployment Configuration

Type: Documentation

Use this folder for configuration references that operators and deployers need, such as required environment variables, connection strings, and service-specific settings (for example .NET or Java services).

Add new docs here when a service introduces new configuration keys or deployment-time requirements.

Configuring .NET Services

Type: Documentation

Swagger

Property	Description	Required	Default Value	Secret?
EnableSwagger	Enable Swagger spec and UI	No	false	No

Azure App Config Environment Variables

The following should be specified as environment variables when launching the service:

Name	Description	Required	Default Value	Secret?
ExternalConfigurationSource	Specifies the configuration source to use	Yes	None	No
ConnectionStrings__AzureAppConfiguration	Connection string for Azure App Configuration	Yes (if using Azure)	None	Yes

Blob Storage

Name	Description	Required	Default Value	Secret?
InternalBlobStorage__ConnectionString	Connection string for internal ABS	Yes (if using ABS)	None	Yes
InternalBlobStorage__BlobContainerName	Blob container name for internal ABS	Yes (if using ABS)	None	No
InternalBlobStorage__BlobRoot	Blob root (name prefix) for internal ABS	No	None	No

ExternalBlobStorage__ConnectionString	Connection string for external ABS	No	None	Yes
ExternalBlobStorage__BlobContainerName	Blob container name for external ABS	Yes (if using ABS)	None	No
ExternalBlobStorage__BlobRoot	Blob root (name prefix) for external ABS	Yes (if using ABS)	None	No

Kafka

Name	Description	Required	Default Value	Secret?
KafkaConnection__BootstrapServers__0	Kafka broker address	Yes	None	No
KafkaConnection__GroupId	Consumer group identifier	Yes	None	No
KafkaConnection__ClientId	Unique client identifier	Yes	None	No
KafkaConnection__SaslProtocolEnabled	Whether to enable SASL auth	No	false	No
KafkaConnection__Mechanism	SASL mechanism	No	Plain	No
KafkaConnection__Protocol	SASL protocol	No	SaslPlaintext	No

Note: Mechanism and Protocol are only used when SASL authentication is enabled.

Options for Mechanism:

- **Plain**
- Gssapi
- ScramSHA256
- ScramSHA512
- OAuthBearer

Options for Protocol:

- **SaslPlaintext**
- Plaintext
- Ssl
- SaslSsl

SASL Plain Text Authentication

Include the following configuration properties to connecto to Kafka that requires SASL_PLAINTEXT authentication:

Name	Value	Secret?
KafkaConnection__SaslProtocolEnabled	true	No
KafkaConnection__SaslUsername	<username>	No
KafkaConnection__SaslPassword	<password>	Yes

Kafka Consumer Settings

Name	Description	Secret?
ConsumerSettings__ConsumerRetryDuration	A list of strings representing retry duration intervals for consumers	No
ConsumerSettings__DisableRetryConsumer	A boolean flag to enable/disable the retry mechanism for consumers	No
ConsumerSettings__DisableConsumer	A boolean flag to completely enable/disable the consumer	No

CORS

Property	Description	Required	Default	Secret?
CORS__AllowAllOrigins	Allow all origins	No	false	No
CORS__AllowedOrigins	Array of allowed origins	No	[]	No

CORS__AllowAllMethods	Allow all HTTP methods	No	true	No
CORS__AllowedMethods	Array of allowed HTTP methods	No	["GET", "POST", "PUT", "DELETE", "OPTIONS"]	No
CORS__AllowAllHeaders	Allow all headers	No	true	No
CORS__MaxAge	Preflight cache duration (seconds)	No	600	No

Token Service Settings

TODO: Describe what the purpose of the token endpoint is

Property	Type	Description	Default Value	Secret?
LinkTokenService__EnableTokenGenerationEndpoint	bool	Controls whether the token generation endpoint is enabled	false	No
LinkTokenService__Authority	string	The authority URL used for token validation/generation	Required (no default)	No
LinkTokenService__LinkAdminEmail	string?	Email address for Link administration	null	No
LinkTokenService__TokenLifespan	int	The lifespan of generated tokens in minutes	10	No
LinkTokenService__SigningKey	string?	The key used for signing tokens	null	Yes
LinkTokenService__LogToken	bool	Controls whether token logging is enabled	false	No

Service Authentication

Property	Description	Required	Default Value	Secret?
Authentication__EnableAnonymousAccess	Enable anonymous access to the service	No	false	No
Authentication__Schemas__LinkBearer__Authority	Authority URL for Link Bearer authentication	Yes	None	No
Authentication__Schemas__LinkBearer__ValidateToken	Validate the token on each request	No	true	No
DataProtection__Enabled	Enable data protection for sensitive data	No	false	No

Redis

Name	Description	Required	Secret?
ConnectionStrings__Redis	Redis connection string	Yes (if caching enabled)	Yes
Redis__Password	Redis Password	Yes (if caching enabled)	Yes
Cache__Enabled	Toggle for Redis caching	No	No

Telemetry

Name	Description	Required	Secret?
Telemetry__EnableTelemetry	Enable Telemetry	Yes (bool)	No
Telemetry__EnableTracing	Enable tracing	Yes (bool)	No
Telemetry__EnableOtelCollector	Enable Otel collector	Yes (bool)	No
Telemetry__OtelCollectorEndpoint	Otel collector endpoint	Yes (uri)	No

Service Registry

The Service Registry section contains URLs and configurations for all microservices in the Link Cloud ecosystem. Configuration key: ServiceRegistry

Not all services use every URL. The configuration is provided for completeness and future-proofing. If using Azure App Configuration, all service URLs should be specified, and each service will use only the URLs it needs.

Service URLs

Property	Required	Description
ServiceRegistry__AccountServiceUrl	Yes	Base URL for the Account service
ServiceRegistry__PublicAccountServiceUrl	No	Public base URL for the Account service
ServiceRegistry__AuditServiceUrl	Yes	Base URL for the Audit service
ServiceRegistry__PublicAuditServiceUrl	No	Public base URL for the Audit service
ServiceRegistry__CensusServiceUrl	Yes	Base URL for the Census service
ServiceRegistry__PublicCensusServiceUrl	No	Public base URL for the Census service
ServiceRegistry__DataAcquisitionServiceUrl	Yes	Base URL for the Data Acquisition service
ServiceRegistry__PublicDataAcquisitionServiceUrl	No	Public base URL for the Data Acquisition service
ServiceRegistry__MeasureServiceUrl	Yes	Base URL for the Measure service
ServiceRegistry__PublicMeasureServiceUrl	No	Public base URL for the Measure service
ServiceRegistry__NormalizationServiceUrl	Yes	Base URL for the Normalization service

ServiceRegistry__PublicNormalizationServiceUrl	No	Public base URL for the Normalization service
ServiceRegistry__NotificationServiceUrl	Yes	Base URL for the Notification service
ServiceRegistry__PublicNotificationServiceUrl	No	Public base URL for the Notification service
ServiceRegistry__QueryDispatchServiceUrl	Yes	Base URL for the Query Dispatch service
ServiceRegistry__PublicQueryDispatchServiceUrl	No	Public base URL for the Query Dispatch service
ServiceRegistry__ReportServiceUrl	Yes	Base URL for the Report service
ServiceRegistry__PublicReportServiceUrl	No	Public base URL for the Report service
ServiceRegistry__SubmissionServiceUrl	Yes	Base URL for the Submission service
ServiceRegistry__PublicSubmissionServiceUrl	No	Public base URL for the Submission service
ServiceRegistry__ValidationServiceUrl	Yes	Base URL for the Validation service
ServiceRegistry__PublicValidationServiceUrl	No	Public base URL for the Validation service
ServiceRegistry__TerminologyServiceUrl	Yes	Base URL for the Terminology service
ServiceRegistry__PublicTerminologyServiceUrl	No	Public base URL for the Terminology service
ServiceRegistry__TenantService__TenantServiceUrl	Yes	Base URL for the Tenant service
ServiceRegistry__TenantService__PublicTenantServiceUrl	No	Public base URL for the Tenant service
ServiceRegistry__TenantService__CheckIfTenantExists	No	Whether to validate tenant's existence in other services such as Data Acquisition
ServiceRegistry__TenantService__GetTenantRelativeEndpoint	No	Relative endpoint path for tenant retrieval in other services such as Data Acquisition; excluding "/api"

Example Configuration in JSON

```
{
 "ServiceRegistry": {
  "AccountServiceUrl": "http://localhost:8060",
  "AuditServiceUrl": "http://localhost:8062",
  "CensusServiceUrl": "http://localhost:8064",
  "DataAcquisitionServiceUrl": "http://localhost:8065",
  "MeasureServiceUrl": "http://localhost:8067",
  "NormalizationServiceUrl": "http://localhost:8068",
  "NotificationServiceUrl": "http://localhost:7434",
  "QueryDispatchServiceUrl": "http://localhost:8071",
  "ReportServiceUrl": "http://localhost:8072",
  "SubmissionServiceUrl": "http://localhost:8073",
  "TenantService": {
   "TenantServiceUrl": "http://localhost:8074",
   "CheckIfTenantExists": true,
   "GetTenantRelativeEndpoint": "facility/"
  },
  "ValidationServiceUrl": "http://localhost:8075",
  "TerminologyServiceUrl": "http://localhost:8076"
 }
}
```

Tenant Service Configuration

Property	Description	Required	Default
TenantService__TenantServiceUrl	Base URL for the Tenant service	Yes	None
TenantService__CheckIfTenantExists	Whether to validate tenant existence	No	true
TenantService__GetTenantRelativeEndpoint	Relative endpoint path for tenant retrieval	No	"facility/"
FacilityIdSettings__NumericOnlyFacilityId	Enforce NHSN ORG ID format (5 digits)	No PRD ONLY	false

Data Acquisition Service Configuration

The following settings are specific to the Data Acquisition service.

Property	Description	Required	Default Value	Secret?

AcquisitionJobSettings__CronSchedule	Cron schedule for the data acquisition background job (e.g., "0/10 * * * * ?" for every 10s)	No	"0/10 * * * * ?"	No

Databases

SQL Server Database

Name	Description	Required	Default Value	Secret?
DatabaseProvider	Database provider to use	No	SqlServer	No
ConnectionStrings__DatabaseConnection	MSSQL connection string	Yes	None	Yes
AutoMigrate	Automatically migrate the database schema	No	false	No

Mongo Database

Name	Description	Required	Default Value	Secret?
MongoDB__ConnectionString	MongoDB connection string	Yes	None	Yes
MongoDB__DatabaseName	MongoDB database name	Yes	None	No

Service Information

The ServiceInformation configuration allows exposing build, version and deployment information through the /api/XXX/info (i.e. /api/audit/info) endpoint of each service. This information is automatically populated from assembly metadata and git information during build time.

Property	Description	Required	Default Value	Secret?
ServiceInformation__ServiceName	Display name of the service	No	Varies service by service	No

ServiceInformation__Version	Version of the service	No	Assembly ver.	No
ServiceInformation__ProductVersion	Version of the product as a whole	No	N/A	No
ServiceInformation__Build	Build number from CI/CD pipeline	No	"dev" when using Docker Compose	No
ServiceInformation__Commit	Git commit hash of deployed version	No	None	No

The service automatically extracts information from .NET assembly version in the AssemblyInfo.cs file or project file

EF Core Query Logging

Configure EF Core to log queries, including sensitive data, by setting up the appsettings.json file and adjusting the Program.cs file. This allows dynamic adjustment of logging levels in Serilog based on the EnableEnhancedQueryLogging setting.

Configuration in appsettings.json

Add the following section to enable enhanced query logging:

```
"EnhancedQueryLoggingSettings": {
  "EnableEnhancedQueryLogging": false
}
```

Property	Description	Required	Default Value	Secret?
EnhancedQueryLoggingSettings__EnableEnhancedQueryLogging	Flag to enable detailed EF Core query logging (Information level)	No	false	No

Configuration in Program.cs

In Program.cs, configure Serilog to dynamically set log levels for EF Core namespaces and enable EF Core logging in the DbContext setup.

For Serilog configuration (e.g., in a ConfigureLogging method):

```
// Bind the enhanced logging settings
var enhancedLoggingSettings =
builder.Configuration.GetRequiredSection(nameof(EnhancedQueryLoggingSettings)).Get<EnhancedQueryLoggingSettings>();

// Create Serilog logger configuration
var loggerConfig = new LoggerConfiguration()
    .ReadFrom.Configuration(builder.Configuration)
    // ... other enrichers and filters ...

// Conditionally override EF Core log levels
var efCoreLogLevel = enhancedLoggingSettings != null && enhancedLoggingSettings.EnableEnhancedQueryLogging
    ? LogEventLevel.Information
    : LogEventLevel.Warning;

loggerConfig
    .MinimumLevel.Override("Microsoft.EntityFrameworkCore", efCoreLogLevel)
    .MinimumLevel.Override("Microsoft.EntityFrameworkCore.Database.Command", efCoreLogLevel);

Log.Logger = loggerConfig.CreateLogger();
builder.Logging.AddSerilog();
```

For DbContext registration (e.g., when using MongoDB provider for EF Core):

```
builder.Services.AddDbContext<MongoDbContext>((sp, options) =>
{
    var querySettings =
builder.Configuration.GetRequiredSection(nameof(EnhancedQueryLoggingSettings)).Get<EnhancedQueryLoggingSettings>();

    // ... client and settings setup ...

    if (querySettings != null && querySettings.EnableEnhancedQueryLogging)
    {
        var loggerFactory = sp.GetRequiredService<ILoggerFactory>();
        options.UseMongoDB(client, mongoSettings.DatabaseName)
            .UseLoggerFactory(loggerFactory)
            .EnableSensitiveDataLogging();
    }
    else
    {
        options.UseMongoDB(client, mongoSettings.DatabaseName);
    }
});
```