SLI – *Technical Specification*

December 12, 2012

SLI – *Technical Specification*

December 12, 2012

The SLI Technical Specification document provides the technical implementation details for the Shared Learning Infrastructure. It contains information on the following topics:

- SLI System Architecture
- Infrastructure Overview
- Data Ingestion
- SLI Data Store
- SLI API
- Application Layer
- SLI Security
- Back-End Operations
- Application Development on the SLI Platform
- Tenant On-boarding Process

SLI System Architecture

This topic provides an overview of the SLI architecture with descriptions of its major design points and subsystems. This specification uses common software design terminology such as concerns, patterns, and loose coupling. The remaining topics throughout this specification cover the implementation details for each component described here.

Design Goals

The SLI design is strongly influenced by the following goals:

- Code reuse
- Separation of concerns
- Loose coupling
- Leveraging open source libraries

The design of SLI is strongly influenced by three major works in the field of patterns:

- *Patterns of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley Professional, 2002)
- *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf (Addison-Wesley Professional, 2003)
- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994)

The sections that follow outline the architecture, integration, and design patterns in SLI.

Architecture patterns

Service Layer
By adding a service layer, SLI allows a coarse-grained access to data elements and promotes abstraction in how data is accessed and stored in its database. SLI uses the service layer pattern in its dashboard application, API, and ingestion system.

Database Session State
By using the database session state pattern, the memory space of each API node is stateless and only the variables needed by an API call are stored in the memory space. SLI uses the database session state pattern so that API nodes will scale horizontally.

Model-view-controller (MVC)
By implementing the MVC pattern, the data model, presentation, and logic that go into the software are loosely coupled and can evolve separately as the system evolves. SLI uses the MVC pattern in its dashboard and other applications.

Integration patterns

Messaging
By utilizing messaging as an integration pattern, machines with contrasting local software can connect and communicate, and the systems can delegate large tasks to multiple machines for scalability. This also contributes to the broad goal for loose coupling. SLI uses messaging services between its various components.

Publish-subscribe and Point-to-point Channels
By using a publish-subscribe channel for message routing, a machine can broadcast messages to multiple possible listeners. SLI uses a publish-subscribe channel pattern for its ingestion system. By using a point-to-point channel, a machine can allow multiple listeners to take messages from a queue so that only one action is executed per message. SLI uses a point-to-point channel pattern for its ingestion system.

Content-based Router
By exchanging messages using a content-based router, a system can have various types of messages in the same message queue. SLI uses a content-based router in the ingestion system, which determines how to route each message based on the content of its header.

Design patterns

Strategy (behavioral)
By using a strategy pattern, software can allow its interface to evolve without exposing its implementation details. This is another step toward loose coupling. SLI uses the strategy pattern in its own client code and facilitates the strategy pattern for third-party application developers.

Proxy (structural)
By using a proxy pattern, software handles objects and data in a way that addresses cross-cutting concerns around security, caching, logging,

and metrics gathering. SLI uses the proxy pattern to allow the software to evolve without the overhead of tedious, unmaintainable inter-dependencies.

Abstract Factory (creational)
By using an abstract factory pattern, software defines a series of interfaces for general usage while hiding the implementation details of its core classes. SLI uses the abstract factory pattern in cases where multiple different implementations need to exist for each general use case, such as in ingestion handling.

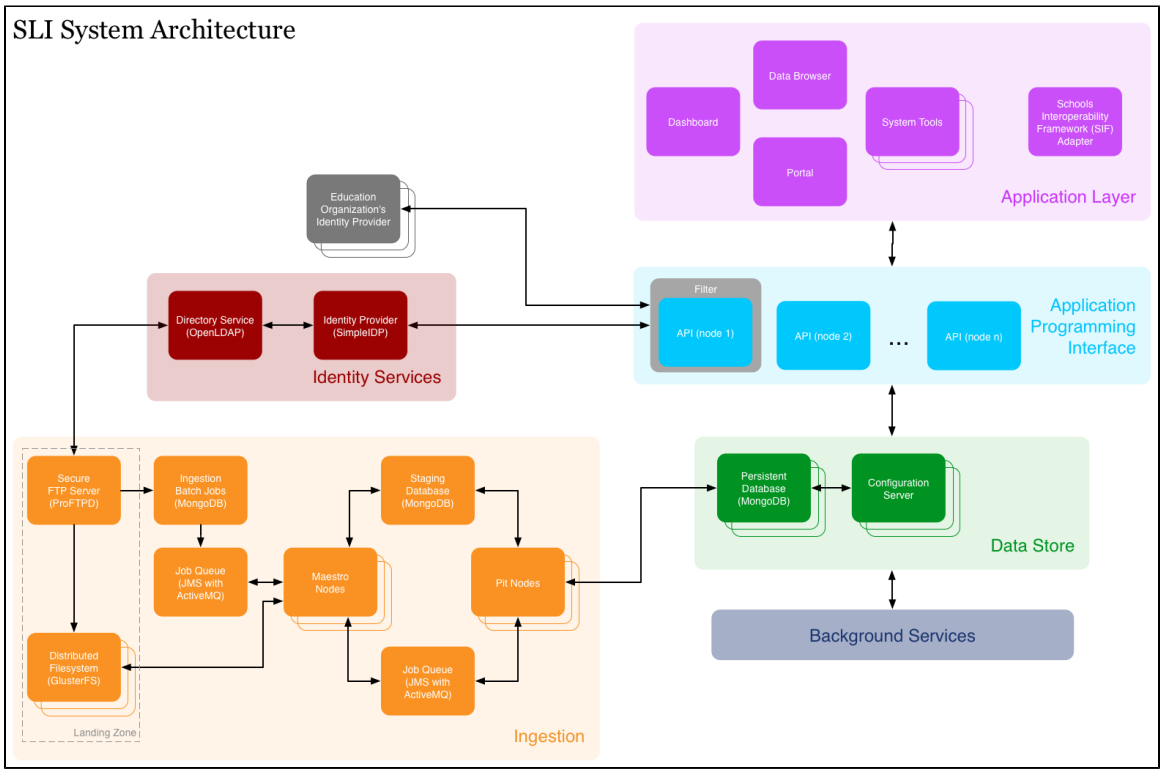
User Roles Considered During Design

Each of the following user roles were taken into consideration during SLI design and development:

- Operators responsible for deploying SLI for an education organization
- Technical staff for an education agency, responsible for supporting SLI
- An education agency's SLI administrators, responsible for managing users and applications for an education agency
- Educators, such as school faculty adding and updating data on a small scale through application interfaces
- School data managers who need to add and update large amounts of data through bulk operations
- Third-party software developers creating applications for use on the SLI platform

Infrastructure Overview

The diagram below is a visual representation of the components and connections throughout the Shared Learning Infrastructure. Each box represents either a logical or physical partition of the code or the concern that the component is trying to address. All communication points are done over secure transport because the system is deployed on an Amazon Virtual Private Cloud.



Major Subsystems

As shown in the architecture diagram above, SLI components fall into subsystems, shown as different color groupings. Each subsystem addresses a specific group of concerns. The following sections provide a broad definition of each subsystem and its components and connections.

Ingestion Subsystem

The ingestion subsystem processes large XML files and saves the data in those files to the SLI data store. This procedure has been called "bulk data ingestion" and "data integration," but this specification uses the term "ingestion." The ingestion subsystem is designed to ensure redundancy and to readily scale for bulk data loads.

The primary component of ingestion is a Java web application, which operates as a processing service. The ingestion subsystem retrieves files that are posted to a landing zone. Typically, the landing zone consists of an SFTP server (ProFTD) and a distributed filesystem (GlusterFS).

Once the ingestion subsystem receives files in its landing zone, the ingestion work begins. One or more maestro ingestion servers partition the ingestion work to be handled by multiple pit worker nodes. A message broker (ActiveMQ) handles the queues that alert the maestro about incoming jobs and that provide work to the pit nodes.

During data ingestion, SLI uses a deduplication and transformation database to speed data processing and reprocessing efforts. This is called a staging database, and it uses a sharded MongoDB cluster similar to the SLI data store subsystem.

Identity Services Subsystem

SLI includes an identity provider application called SimpleIDP. SimpleIDP and the SLI's directory service (using OpenLDAP) handles authentication for SLI Operators - those who deploy and manage the infrastructure. SimpleIDP also handles application developers using a

sandbox environment.

For administrators and end users, SimpleIDP integrates with an education agency's identity provider (IDP). These users are referred to as federated users. For federated users, SLI determines a user's access rights to data by examining the roles that user has in the agency's IDP and the role mappings configured by SLI administrators.

Applications rely on identity services for signing in and obtaining security tokens for asynchronous database transactions. Identity services use Security Assertion Markup Language (SAML) over HTTPS, and applications obtain and use OAuth authorization tokens through the API.

Data Store Subsystem

The core data store uses the MongoDB database software. SLI has MongoDB databases to store core education data and system-level configuration data. Within the subsystem, MongoDB databases use sharding as a horizontal partitioning approach, creating a series of nodes. Mongo configuration servers support the partitioned structure. The nodes are divided into replica sets. For communication with other SLI subsystems, the data store uses the Mongo Wire Protocol over TCP/IP.

Application Programming Interface (API)

SLI uses the Representational State Transfer (REST) architectural approach for its application programming interface (API). By using REST, SLI provides its API as a series of stateless Java nodes that respond to HTTP requests and provide authenticated, authorized access to data from the SLI data store.

Applications that make API calls must start each user's session with an OAuth 2.0 authorization request. For each HTTP request to the API, an API filter inspects the request for an OAuth 2.0 token. If that security token is not present, the API redirects the request to an identity provider (IDP). All communication with an IDP uses the SAML 2.0 protocol.

Application Layer

The application layer features a group of interactive web applications for SLI users. Each application has its own target audience, such as system tools for SLI administrators and a customizable dashboard application for educators. Also in this layer is the Schools Interoperability Framework (SIF) adapter for other applications developed to work on the SLI platform. Open source versions of the default web applications offer developers examples to work from.

The following sections define the default applications in this layer.

Portal

The portal application is the central point of access for all SLI users. The portal is a Java web application based on the Liferay portal platform. Depending on portal configuration within a specific SLI deployment, the portal provides links to other web applications that the current user has access to. For administrators and operators, this includes an "Admin" link to access system tools.

Dashboard

The dashboard is a Java web application designed for end-users to view data to which they have access. The dashboard retrieves data by calling the REST API. Using a separate dashboard builder tool, administrators can customize the dashboard experience for non-administrator users.

Data Browser

The data browser is a Ruby on Rails application designed for IT administrators to view and verify ingested data. The data browser retrieves data by calling the REST API, and it presents HATEOAS links from the HTTP responses for the purpose of navigating the data by its relationships. The data browser application is also available to application developers to aid them in managing their sandbox environments.

System Tools

The system tools are a single Ruby on Rails application for high-level platform management. This includes creating landing zones for ingestion and managed realm users and registered applications. Users with administrative permissions in the SLI can access system tools from an "Admin" link in the SLI portal.

Supporting Technology

SLI employs several commercial software applications, protocols, and services within the infrastructure. The following is a list of those supporting technologies along with a brief description of how they are used in the SLI.

Red Hat Enterprise Linux

Red Hat Enterprise Linux is a Linux operating system geared towards commercial and enterprise deployments. SLI was developed for deployment on Red Hat Enterprise Linux 6. For more information, see http://www.redhat.com/.

MongoDB

MongoDB is a NoSQL document database platform by 10gen (http://www.10gen.com). SLI uses MongoDB for record storage and ingestion data transformation. For more information, see http://www.mongodb.org/.

OpenLDAP

OpenLDAP is an Open Source Lightweight Directory Access Protocol implementation. SLI uses OpenLDAP in SimpleIDP, system tools, and landing zone platform components to store information about administrative realm users. For more information, see http://www.openldap.org/.

Java

Java is an compiled bytecode interpreted language originally developed by Sun Microsystems and now maintained by Oracle. Since its release, software developers worldwide have established a variety of Java implementations. SLI was developed using the Oracle Java Development Kit, and much of the Java-based substrate requires the use of Oracle's Java distribution. For more information, see http://www.oracle.com/.

Apache Tomcat

Apache Tomcat is an open source application server for hosting Java-based web applications. SLI uses Tomcat to host its REST API, ingestion components, and Java-based applications (dashboard and portal). For more information, see http://tomcat.apache.org/.

Apache ActiveMQ

Apache ActiveMQ is a messaging platform that's compliant with the Java Messaging Service (JMS) protocol and compatible with Streaming Text Oriented Messaging Protocol (STOMP). SLI uses ActiveMQ for both JMS and STOMP message queues. These queues can be found in the ingestion subsystem. ActiveMQ is intended to be deployed in a cluster. For more information, see http://activemq.apache.org/.

Ruby

Ruby is an interpreted programming language. SLI system tools and data browser are written in Ruby, as are several utilities and scripts used for managing SLI. For more information, see http://www.ruby-lang.org/.

Ruby on Rails

Ruby on Rails (RoR) is a web application framework written in the Ruby programming language and based upon the Rails framework. In production, RoR applications typically run via an open source tool called Passenger which is attached to the web server of choice (often Apache HTTPd or Nginx). For more information, see http://rubyonrails.org/ and https://www.phusionpassenger.com/.

Apache HTTPd or Nginx Web Servers

In SLI, Apache HTTPd or Nginx Web Servers can be used for providing a solidified platform for serving Ruby on Rails applications, as well as performing pass-through to back-end components if deemed necessary. For more information, see http://httpd.apache.org/ and http://nginx.org/.

Liferay

Liferay is an open source web portal application. SLI bases its portal application on Liferay, creating the user's primary landing page for access to SLI applications and tools. For more information, see http://www.liferay.com/.

MySQL

MySQL is an open source relational database management system. SLI uses MySQL to support the portal application: for portal-specific data storage and for sharing data between portal servers. For more information, see http://www.mysql.com/.

GlusterFS

GlusterFS is an open source distributed file system technology provided by Red Hat, Inc. This technology allows for a scalable distributed file storage area that can be shared between a number of servers. SLI uses GlusterFS in the ingestion subsystem, serving as the destination for files uploaded for ingestion processing. For more information, see http://www.gluster.org/.

ProFTPD

ProFTPD is a highly configurable open source FTP server which includes support for SSH File Transfer Protocol (SFTP) as well as post-upload program execution. SLI uses ProFTPD to the landing zone, allowing remote authenticated users to upload files for ingestion processing. For more information, see http://www.proftpd.org/.

ElasticSearch

ElasticSearch is a Java-based distributed-index search platform that allows free-form search queries to be executed against its database. SLI uses ElasticSearch to support free-form search queries. In SLI, ElasticSearch is fed data via the SARJE processing jobs. For more information, see http://www.elasticsearch.org/. As of this writing, ElasticSearch is not yet implemented in SLI but is scheduled for an upcoming release.

Apache Hadoop

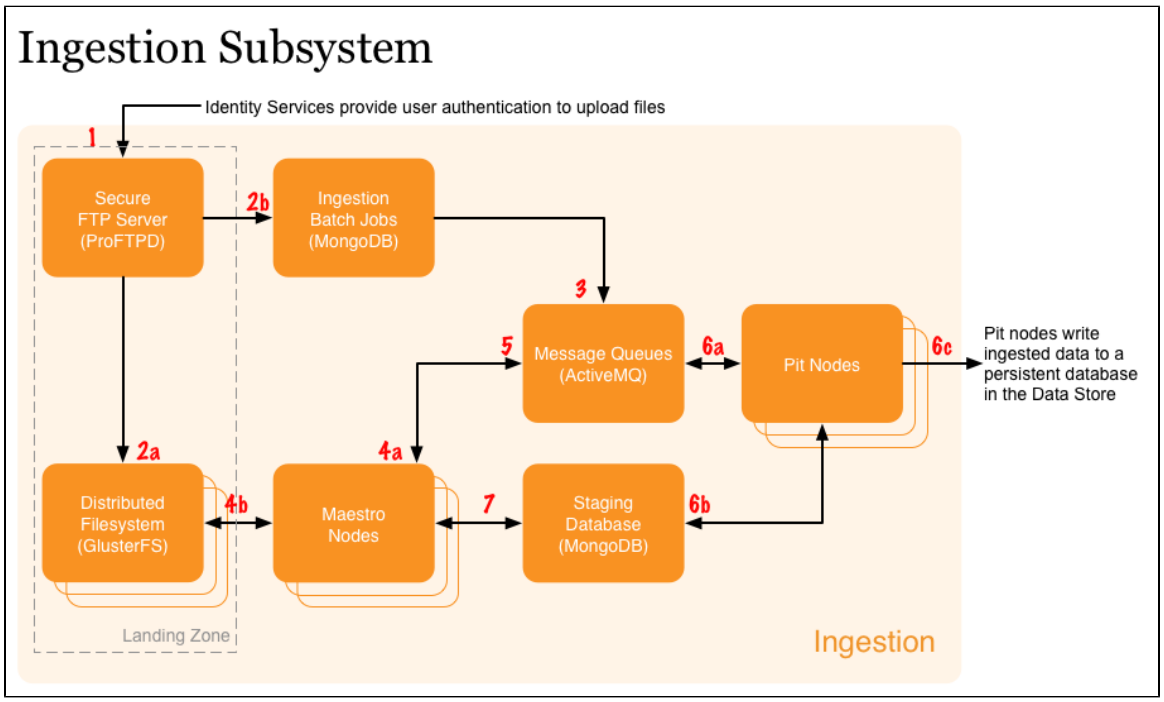
Apache Hadoop is a Java-based framework for reliable, distributed, scaled data processing. In SLI, SARJE uses Hadoop for aggregation processing. For more information, see http://hadoop.apache.org/. As of this writing, Hadoop is not yet implemented in SLI but is scheduled for an upcoming release.

Data Ingestion

Subsystem Overview

Ingestion is the process by which data is uploaded to SLI through a series of bulk operations. Through ingestion, users can add new data or perform a large series of updates on existing data. Ingested data must be uploaded as a set of one or more properly formatted XML files, usually as a compressed archive (.ZIP). These XML files must confirm to the SLI-EdFi schema, which is the Ed-Fi schema plus extensions.

The following diagram and tables introduce the components and connections in the ingestion subsystem.



Components

Component	Description	SLI Role
ProFTPD	Secure FTP server	Receives files containing the data to be ingested.
GlusterFS	Distributed filesystem	Stores ingested files, replicating data for access from many remote nodes.
Maestro	Java-based master processing node	Receives the incoming file, parses the file, and partitions it for further processing by pit nodes. Also responsible for aggregating job results at the end of a job.
Pit	Java-based worker node	Processes data during the ingestion process after it has been staged and partitioned. Responsible for executing integration steps of transformation and persistence.
MongoDB (Staging)	Database	An intermediary database used to store data while it is still processing for ingestion, prior to saving it to the core database.
ActiveMQ	Message broker	Server with the message queues for ingestion, including the incoming ingestion batch jobs (using STOMP) and the maestro's partitions to be handled by pit nodes (using JMS)
MongoDB (Ingestion Batch Job)	Database	An information database that is used to store ingestion errors, job metrics, job intermediary state, and other job-related information.

Connections

Connection	Description
1	Authentication via PAM. ProFTPD authenticates users against the Open LDAP directory service.
2a	ProFTPD writes incoming files to a GlusterFS mount.
2b	Upload traffic triggers ingestion batch jobs by posting a message to the ActiveMQ message broker.
3	Ingestion batch jobs are queued for processing by one or more maestro nodes.
4a	Maestro takes an ingestion batch job off the queue.
4b	Maestro reads incoming files from GlusterFS, and writes outbound log and error reports to that mount.
5	Maestro farms out work items for pit nodes by adding them to an ActiveMQ job queue, and it listens and responds to the messages received back from the pit nodes.*
6a	Pit takes an ingestion work item off the queue and performs ingestion processing.*
6b, 6c	Pit adds its results to the staging database, and to the persistent database for the ingestion user's tenancy.
7	Maestro interacts with the staging database based on how the pit nodes are performing tasks from the job queue.

* In its version 1.0 release, SLI supports running in standalone mode only, which means that the maestro and pit nodes are not separated, and a single ingestion node performs all the tasks specified for both maestro and pit nodes.

Multitenancy

The Ingestion Subsystem diagram shows the components as if there is a single tenancy with a single landing zone. However, in production a single SLI could serve multiple tenancies, each with its own tenant database. Also, each tenancy could have one or more landing zones created to serve the ingestion processes to its database.

In production, a single tenancy typically represents an *education agency* within that SLI, either local (*LEA*) or state (*SEA*).

Pre-ingestion Preparation

Before ingesting data for a tenancy, users must take steps to prepare their accounts, landing zones, and data as described in the following sections.

Landing Zone Creation

Part of tenant on-boarding includes creating one or more landing zones used to ingest data for that tenancy. To do this, an SLI administrator can use the web-based administrator tool for managing landing zones. Once a landing zone is ready for use, the tool lists the SFTP server information that ingestion users need for uploading files. Additional landing zones for a tenancy can be created as needed. For details about the landing zone specification, see Landing Zone.

Ingestion User Registration

During and after tenant on-boarding, an SLI administrator can create and manage SLI users with the *ingestion user* role for that tenancy. This means that the user can authenticate with SLI and upload files to any landing zone associated with that tenancy. A user supplies these SLI credentials when uploading files to the landing zone (specifically to the SFTP server).

File Formats for Ingestion

There are three of file formats to consider when preparing data for ingestion:

- The XML files that contain the data to upload become part of the archive file. These files must conform to the SLI ingestion schema described in the sections that follow.
- A control file is packaged with the XML files in the archive. An upcoming section describes this control file and how the ingestion process makes use of it.
- The file to upload to the landing zone must be in the zip file format (with the **.zip** extension). During pre-ingestion preparation, a user must use an archive tool to create this industry standard archive format. The archive should contain the XML files and control file for an ingestion batch operation.

All files are assumed to be UTF-8 encoded unless otherwise specified. Machine-readable timestamps use Unix time (UTC), and human-readable timestamps use a formatted date/time string.

XML Files in the Ingestion Schema

The SLI ingestion schema is an extension of the Ed-Fi Core standard (download). The sections that follow describe the construction of this ingestion schema.

Ed-Fi Core

The Ed-Fi Core data standard was created as a unifying logical data model for educational data. The Ed-Fi Core schema aligns with the Common Education Data Standards (CEDS) (ceds.ed.gov). By aligning with Ed-Fi, the SLC ingestion schema also aligns with CEDS. The SLI ingestion schema currently conforms to Ed-Fi Core version 1.0.03. (Download Ed-Fi Core 1.0.03 documentation..)

Interchange Schemas

The Ed-Fi interchange schemas define XML representations of particular data spaces, or groups of entities and associations, for transporting data between systems. Different interchange schemas may be used to reflect different use cases, such as different groups of source systems. Ed-Fi Core (previously defined) provides a library of building blocks, which are referenced from the interchange schemas.

SLI has interchange schemas that are largely based on the interchange schemas defined by Ed-Fi. The SLI schema covers most, but not all, of the entities in the Ed-Fi standard. The SLI internal schema consolidates some of the Ed-Fi entities for the sake of scalability and performance in both the bulk ingestion subsystem and the REST API.

The following are the themes behind the SLI extensions to the Ed-Fi interchange schema:

- Removes ID-REF support
- Constrains entity references to use one pattern of natural keys for referencing entities
- Adds and removes attributes to conform with SLI requirements

For each XML file to be ingested by the SLI, the control file cites the SLI interchange schema associated with that file.

Control File

Each zip file uploaded for ingestion must include a control file. The control file is a text file in which each line specifies the file format, interchange schema, file name, and checksum for a file to be ingested. Within each zip file that is uploaded for ingestion, SLI expects to find this file with the ".ctl" extension included alongside its corresponding XML files.

Ingestion Validation Tool

SLI users can validate the XML files to be ingested using the open source ingestion validation tool. SLC makes this tool available as a free download from its GitHub page (github.com/slcedu). Instructions for using the tool are in the SLI administrator documentation, including how to download and unpack the code and how to run the tool from either a Windows or Linux/UNIX command line.

The ingestion validation tool is a Java-based application designed to detect errors in the XML files prior to starting an ingestion operation. As of this writing, the ingestion validation tool only reports errors in structure adherence to the SLI and Ed-Fi schemas.

Record Count Verification Tool

SLC provides the verification script *jsExpected.js* so that SLI operators can verify that the record count stored in the data store is as expected given the data presented for ingestion.

Conversion

In preparing data for ingestion, a user may start with data in another industry standard format. SLC provides tools to convert data from those standards into valid XML in the SLI ingestion schema. The sections that follow cover the tools currently available for use in preparing data for SLI ingestion.

CSV Files

At its GitHub page (github.com/slcedu), the SLC hosts an open source tool called **csv2xml** that transforms CSV files to the SLI ingestion XML schema. Included in the download are sample CSV files for the student and teacher entities that can be transformed using *csv2xml*.

Documentation for *csv2xml* includes the instructions needed to use the tool and to extend the code to cover additional entities. It also explains the process of regeneration of classes to alisgn with changes in Ed-Fi or SLI schemas.

Schools Interoperability Framework (SIF) Data

For SLI, CPSI, Ltd., is responsible for designing, implementing, deploying, and supporting a SIF agent capable of transforming incoming SIF data into one of the source formats supported by SLI. The implementation of the data collection process uses a methodology that addresses the following:

- Dynamics of data input and extraction - The business process of entering data in school districts' Student Information System (SIS) and in other school applications currently residing at the district level.
- Data collection process - An asynchronous process which will require different data sets to be staged while waiting for the rest of the data sets so as to be able to create valid Ed-Fi CSV or XML files.
- Validation of data - The validation of data to ensure data quality and accuracy prior to sending to the SLI collection portal.
- Timeliness - The ability to send the data in near-real-time.
- Data standards - Ready compliance to data element changes and additions.
- File submittals - The generation of accurate and consistent data submittal files in XML format.
- Data changes - Two ways to submit changes in data: submitting full data sets or submitting only data changes.
- Transmitting SIF data from each district into the SLI portal via a SIF Agent.

Landing Zone

The term **landing zone** refers to an upload target for files to be ingested to a specific tenant database. An SLI user with the *ingestion_user* role for that tenancy can securely sign in and upload files to that landing zone.

The user connects to the landing zone using SFTP, authenticated by the same LDAP that authenticates other SLI users in the administrator/operator realm. This provides the same unified credentials and centralized authorization used throughout the SLI.

The landing zone also includes GlusterFS, a distributed file system, as the file storage medium between the landing zone servers and ingestion servers. GlusterFS allows for simple scaling in terms of concurrent operations per second. In addition, it minimizes the chance for a storage space issues in the landing zone or ingestion workspaces.

Landing Zone Setup and Management

An SLI administrator has access administrator tools from the SLI portal, including a landing zone management tool. Using that tool, the administrator can set up new landing zones for a tenancy (occasionally called "provisioning" the landing zone). The administrator can also view information about existing landing zones and manage those zones over time.

By using the landing zone tool, the administrator triggers the following events for setting up the landing zone:

- Create the necessary ingestion accounts and directories.
- Email ingestion users instructions to connect to the landing zone's SFTP server.

When creating a landing zone for an application developer sandbox, the ingestion user also has the option to pre-load a sample dataset from SLC. That triggers the additional event of ingesting the selected dataset in the new landing zone.

Ingestion Processing in the Landing Zone

After a zip file is uploaded to the landing zone, the SFTP server sends a message to a processing queue. Multiple ingestion processes in the ingestion subsystem draw from this same queue. One ingestion process consumes the new message and uses its contents to start processing a new job for the uploaded file(s) against the user's tenant datastore. Once the job processing is complete, a job report file is created in the user's landing zone, including any errors or warnings that were encountered. The user is able to view or download these reports from the FTP server.

Ingestion Process

Earlier in this document, architecture diagrams provided a look at the physical components and connections in the ingestion subsystem. This section looks at the overall ingestion process, or workflow, through that subsystem.

The following is a summary of the ingestion process for 1.0, which operates in _standalone mode_, meaning that all node operations are performed by a single ingestion node:

1. Ingestion processing begins when a file is uploaded, via SFTP, to a tenant's landing zone.
2. When the upload is complete, the FTP server creates a message with information about the uploaded file and sends it to the processing queue. One or more ingestion nodes consume ingestion jobs from that same queue.
3. The ingestion node consumes the new processing message and validates that the path to the file it describes is a valid landing zone in

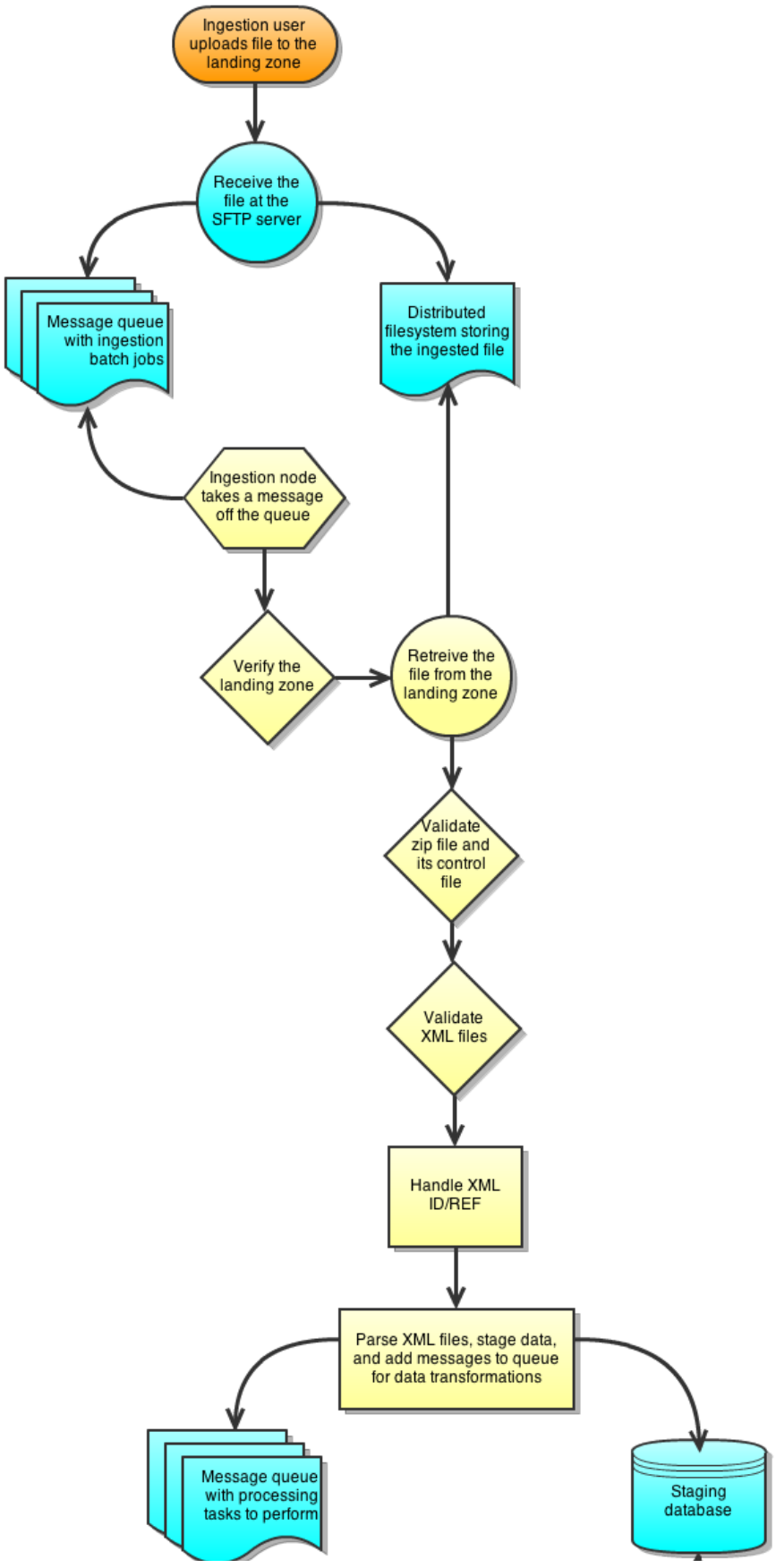
SLI.

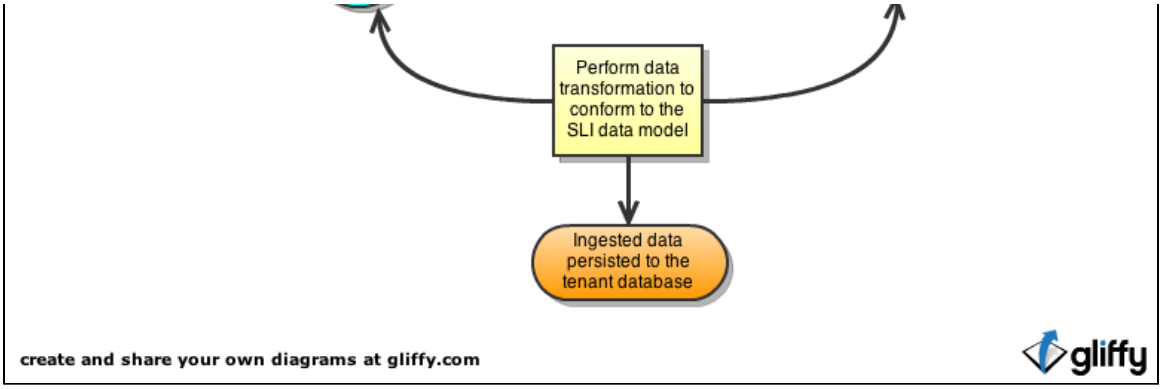
4. If the file path is valid, the ingestion node parses and validates files as follows:
    a. Parse and validate the zip file.
    b. Extract, parse, and validate the control file.
    c. Validate the XML and confirm that ID/REF references are resolved.
5. The ingestion node parses the XML files into objects and stores them in a staging database while using a message queue to orchestrate ingestion processing across multiple nodes.
6. An ingestion node takes a message off the queue and performs part of the ingestion process.
7. The ingestion node saves data to the staging database if additional processing will follow or to the persistent tenant database if the processing is complete.
8. The ingestion node continues driving the process through messages until all ingested data is persisted to the tenant database.
9. Log files containing errors, warnings, and/or statistics and other status information, for the job as a whole, as well as individual XML files, are deposited back on to the landing zone.

The graphic below reflects the workflow described here.

Ingestion Process Flow

Standalone Mode





Maestro and Pit Nodes

The ingestion subsystem uses a series of nodes (servers dedicated to processing tasks) to handle the load of ingesting data after it appears in the landing zone. The ingestion specification for SLI distinguishes between two types of ingestion nodes based on their roles in the ingestion process:

- A **mastro node** processes the files to be ingested, including validation tasks, and saves data to the staging database. Then, it orchestrates the distribution of work to pit/worker nodes (using a message queue), and it performs post-ingestion cleanup and reporting.
- A **pit node** picks up messages from the messaging queue and process data in the staging database in accordance with those messages. This processing transforms the data so that it conforms with the SLI data model. Then, it saves it back to the staging database if it needs further processing, or it saves it to the persistent tenant database associated with the ingestion user.

For SLI version 1.0, this specification is not fully implemented. Instead, the tasks that would be divided between maestro and pit nodes are, instead, handled by a single ingestion node.

Staging Database

The staging database in the ingestion subsystem is an intermediate database that is used to prepare the source data for the SLI database. As maestro nodes parse the incoming XML files, the records are saved to the staging database. These staging database records are referred to as *neutral records*. The pit nodes process the neutral records through data model transformations so they will conform to the SLI data model. As further processing is necessary, the pit nodes then save these records back to the staging database as transformed records.

Validation

The SLI ingestion process performs the following the following validations on the input data:

- Zip file validation - Confirms that the provided input file is a valid zip archive file and includes a control file within the archive.
- Control file validation - Confirms that the entries in the control file are valid by checking that the file exists in the landing zone, is readable, and that the checksum, file format, file type are correct.
- XSD validation - Using an embedded Xerces XML parser, confirms that incoming files validate against SLI-Ed-Fi schemas. For information about the errors and warnings produced by the parser, see Xerces XML parser documentation at xerces.apache.org.

The ingestion process writes the output of zip and control file validation to the job error log. The XML validation from Xerces is written to resource-specific log files.

Persisting Data to the Tenant Database

The persistence step reads the staged entities from the staging database and stores them in the persistent database in the SLI data store.

Ingestion Client Batch Tool

The Ingestion Client Batch Tool is a Java-based application to assist users in uploading data for ingestion. It is available as *batch_upload_tool* from the SLC's GitHub page (github.com/slcedu). The application is command-line-based Windows executable file `ingestion-upload-tool.exe`, so no installation is required, and users can use the Windows Task Scheduler application to run the tool at regular intervals.

What It Does

The Ingestion Client Batch Tool handles the XML data files and associated control files. All files must be located in a single local directory. The

application assumes these files have been prepared for ingestion. It does not perform any additional validation or transformation steps, and it does not track ingestion results.

Upon execution, the tool will package the Ed-Fi data (files ending in .xml) and the control file (file ending in .ctl) into a zip file, connect to the landing zone's SFTP server, and upload the zip file for ingestion into the user's root directory.

Configuration/execution Requirements

The following items are required from the user in order to run the application:

- SFTP server - IP address/URL of the SFTP server that the designated landing zone uses
- SFTP port - SFTP server listening port
- user - user name of an ingestion user account in SLI (associated with the target tenant database)
- password - password for that ingestion user account
- local directory - the absolute/relative path to the local directory where the user's data files and control file are located

System Requirements

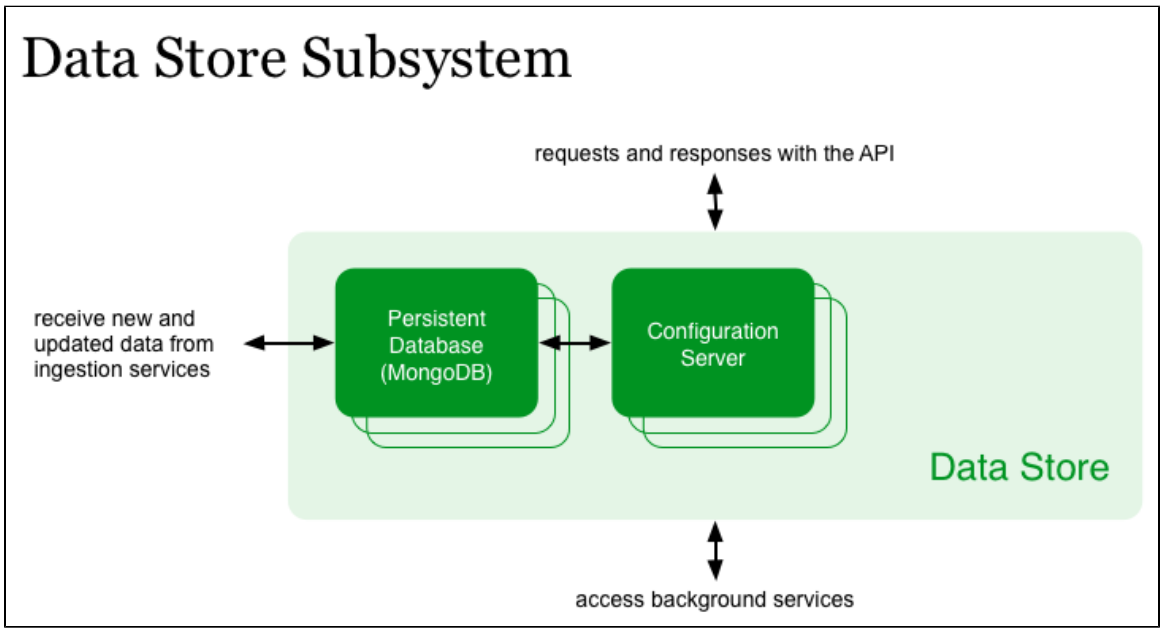
- Microsoft Windows 7
- Java Runtime Environment 1.6
- 512 MB available memory
- 50 MB available disk space
- Internet access

SLI Data Store

Subsystem Overview

SLI uses databases to store core education data, custom educationally-relevant data, and system-level configuration data. The infrastructure features the MongoDB database software and supporting Mongo services and protocols. The data store subsystem includes the persistent Mongo databases and configuration servers.

The following diagram and tables introduce the components and connections in the data store subsystem.



Components

Component	Description	SLI Role
MongoDB	Database	Data storage for system operations and for school data. In SLI, this database is divided into three or more shards and grouped in replica sets of three
Configuration Server	Linux server with Mongo configuration resources	Configuration source for the Mongo cluster, used by *mongos* (the Mongo routing process) and *mongod* (the Mongo database process)

Connections

Connection	Description
All connections to MongoDB databases	All interactions involving a Mongo node use the Mongo Wire Protocol over TCP/IP

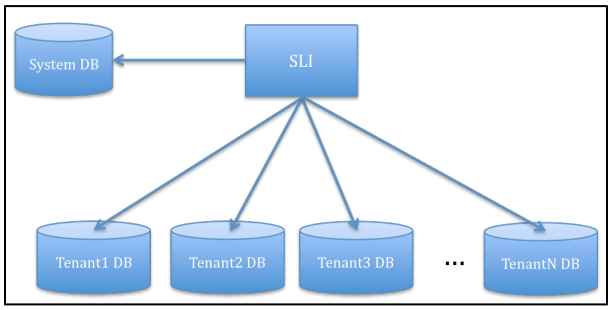
SLI and Tenant Databases

The SLI platform uses a database for all system-specific data (the `sli` database), as well as a database for each tenant's data. In addition to taking advantage of database-level locking, this adds an extra level of security to data access across tenants.

Tenant-level Database Partitioning

Data partitioning is a mechanism to organize data to overcome limitations of the Mongo database in order to achieve better scalability and performance.

SLI partitions data in a way that each tenant's data are contained within a dedicated database. The tenant database and its associated set of search indexes are created during the tenant on-boarding process. This tenant-level separation of data simplifies and optimizes tenant-level operations, which reduces cross-tenant performance degradation. A diagrammatic representation of this structure is depicted below. Currently, the number of possible tenant databases is unbounded.



SLI maintains a separate connection to each database belonging to a tenant during the course of an ingestion job for that tenant. Tenant data are maintained within each tenant database until a purge operation is performed by the tenant.

SLI stores other tenant-independent information in a separate system database, including tenant database info and system-wide data.

Sharding

Sharding a MongoDB collection allows for scaling by spreading data, and therefore read and write operations, across available shards. Shards are essentially partitions of data. The shard key determines how the data in a collection is partitioned among the shards in the cluster. The shard key used in the SLI platform is the `_id` field in each collection. Since SLI IDs are hashes of values contained within the document, the shard key distribution is essentially random. This leads to ideal performance during bulk data ingestion, but sub-optimal performance during reads as no data locality is available.

The `_id` generation can be modified in order to give data locality, in which documents that are frequently accessed together are stored closely.

The `sli` database is not sharded. The SLI system handles sharding each tenant database on creation.

Pre-splitting

When sharding a MongoDB collection, the data can be split in to smaller chunks and those chunks moved to specific shards at any point. The balancer normally handles this operation to keep data balanced among all available shards. However, two issues necessitated turning off the balancer and splitting the chunks programatically, also known as pre-splitting. The first problem is a known MongoDB issue related to counts being inaccurate when a query involves data in the chunk being moved.

The other issue is related to the performance of the ingestion process. During initial onboarding, data is ingested into an empty database. Normally, each collection would be contained within one chunk on one shard. Thus, all ingested data would be ingested onto a single shard. As the data size grows, that shard will split the data into chunks and begin migrating them to other shards. The heavy write load with simultaneous chunk migrations severely degrades the performance of the single shard and can negatively impact the performance of other tenants.

Pre-splitting data and disabling the balancer allows the SLI platform to spread writes among all shards during ingestion and limit the performance impact of heavy write loads. The range of possible shard key values is split in to chunks according to the available number of shards, and those chunks spread evenly among shards. With the balancer off, chunks will never be migrated, and those ranges of data will live permanently on specific shards. A random shard key (see above) spreads reads/writes evenly among each range, which are on different shards. Thus, the load is evenly distributed.

Pre-splitting is not applicable to the `sli` database since it is not sharded. The SLI system handles pre-splitting each tenant database on creation.

Indexing

MongoDB is a document-oriented database, in which each document is a JSON/BSON object. Fields within each document can be indexed to allow for quicker retrieval. The SLI platform indexes each collection in the `sli` database and each tenant database according to querying patterns used by the system.

Additional querying patterns may be added at the expense of more indexes, which degrade write performance and require more shards to keep data in memory.

Indexing the `sli` database must be done during deployment. Since tenant databases are dynamic, the SLI system handles indexing each tenant

database on creation.

Expanding the MongoDB Cluster

In order to increase scalability, shards must be added to the MongoDB cluster. This presents a problem due to the use of pre-splitting. With the balancer off, no data would be migrated to the new shard. In addition, chunks must be migrated in such a way as to maintain an equal, but not necessarily contiguous, range on each shard. The default balancing algorithm always moves the lowest-valued chunk off the shard if a balancing operation is deemed necessary. Thus, the balancer can be turned on temporarily to add a new shard to the cluster. An equal distribution of data among all shards would result in the lowest range from each shard being migrated to the new shard, giving each shard an equal amount of possible shard key values. This process will maintain the random distribution of data.

The bulk data ingestion process relies on counts being accurate, as does paging headers in the API. Thus, a maintenance window must be used to enable the balancer to migrate chunks. Chunk migration is a slow process, but it can be done incrementally until complete.

Deterministic IDs

Every entity persisted into the data store has a unique id. Deterministic ids (DIDs) are identifiers that are generated by applying the SHA1 hash algorithm to a string comprised of the entity's collection name, tenant, and natural keys (with a separator of |~ between all values) - and then appending "_id" to the end of the generated hash.

For example, some sample strings to which the SHA1 hash algorithm would be applied to generate DIDs:

educationOrganization	~Hyrule	~1000000111	~ teacherSectionAssociation	~Hyrule	~c1cdfa54ed19ba1552fe4bdf73a745c6d8851e3e_id

Note that the natural keys may be values or DIDs for other entities, as is the case for references. In the third example above, the **Section** entity has the natural keys of the uniqueSectionCode (Art001-Sec2) and the schoolId (7d0ad460285058e4f924acbd10d416fa3e8dd60e_id).

Deterministic IDs and SuperDocs

With the advent of SuperDocs, some entities have become embedded within other entities. For example, **StudentSectionAssociation** now is embedded within the parent entity of **Section**. The ids for references to these embedded entities will be a concatenation of the parent entity DID and the child entity DID. For example:

StudentGradebookEntry has a reference to a **StudentSectionAssociation**, which is a studentSectionAssociationId of the form <Section DID><StudentSectionAssociation DID>.

From some of our sample data for a **StudentGradebookEntry**:

body.studentSectionAssociationId	135963f2abd3320ae508546fbff31f37e10b949e_id9c8a94ce380481ecda4e95068f5bb69a29da3c58_id
Section DID	135963f2abd3320ae508546fbff31f37e10b949e_id
StudentSectionAssociation DID	9c8a94ce380481ecda4e95068f5bb69a29da3c58_id

The following text explains how to determine the natural keys for an entity.

Natural Keys

The natural keys for an entity can be found in the SLI schema, specifically in ComplexTypes.xsd. Look for the elements that have the annotations of natural keys, i.e. for the **Section** entity one of the natural keys is the unique section code.

```
<xs:element name="uniqueSectionCode" type="uniqueSectionCode"> <xs:annotation> <xs:documentation>...</xs:documentation>
<xs:appinfo> <sli:naturalKey>true</sli:naturalKey> </xs:appinfo> </xs:annotation></xs:element>
```

Make sure also that the entity is annotated to apply the natural keys. For the **Section** example again:

```
<xs:complexType name="section"> <xs:annotation> <xs:documentation>...</xs:documentation> <xs:appinfo>
<sli:CollectionType>section</sli:CollectionType> <sli:applyNaturalKeys>true</sli:applyNaturalKeys>
<sli:schemaVersion>1</sli:schemaVersion> </xs:appinfo> </xs:annotation> <xs:sequence> ...
```

The natural keys for an entity uniquely define that entity and can be used to refer to that entity, therefore enable natural key reference resolution to replace ID/IDRef as the method for SLI reference resolution.

Natural Keys Reference Resolution

Being able to calculate a deterministic id for an entity reference removes the necessity to perform a lookup on the data store to obtain an id.

An additional effect of calculated ids is that reference validation is removed, meaning that a reference to an entity can be resolved to a DID, but there is no guarantee made that the referenced entity exists in the data store. It is the ingestion user's responsibility to provide data with valid references.

The following sections contain details on the code components which resolve references.

Key components

`DidSchemaParser` -- responsible for parsing the Ed-Fi extension schema and creating all deterministic id configuration objects
`DeterministicIdResolver` -- responsible for replacing an entity's reference type with calculated deterministic ids

Configuration objects

The configuration objects below are created by the `DidSchemaParser`.

`DidEntityConfig` -- responsible for defining the location of each reference type in the entity and the SLI fields into which the calculated id for each referenceType should be resolved
`DidRefConfig` -- responsible for defining the key fields contained in a reference type, also configures the deterministic hierarchy for nested deterministic ids

Process flow

When the ingestion engine is started, the `DidSchemaParser` parses the Ed-Fi extension schema (SLI-Ed-Fi-Core.xsd). It creates and maintains the`DidEntityConfig` and `DidRefConfig` objects based on the xsd annotations. A `DidEntityConfig` is created for each type of entity that can be ingested (such as a studentSchoolAssociation), and a `DidRefConfig` is created for each reference type (such as a studentReference). Because a reference type looks the same regardless of the entity containing it (a school reference looks the same whether it's within a studentSchoolAssociation entity or a teacherSchoolAssociation entity), these `DidRefConfig` configurations are reusable.

As entities are processed, they are passed through the `DeterministicIdResolver` to transform their reference objects into calculated deterministic ids. When an entity goes through the `DeterministicIdResolver`, the appropriate `DidEntityConfig` is used to determine which references need to be resolved. For each reference type (specified by a `DidRefConfig`, the `DeterministicIdResolver` calculates a deterministic id based on the entity's values for that reference. The entity's reference object is then replaced with the calculated deterministic id.

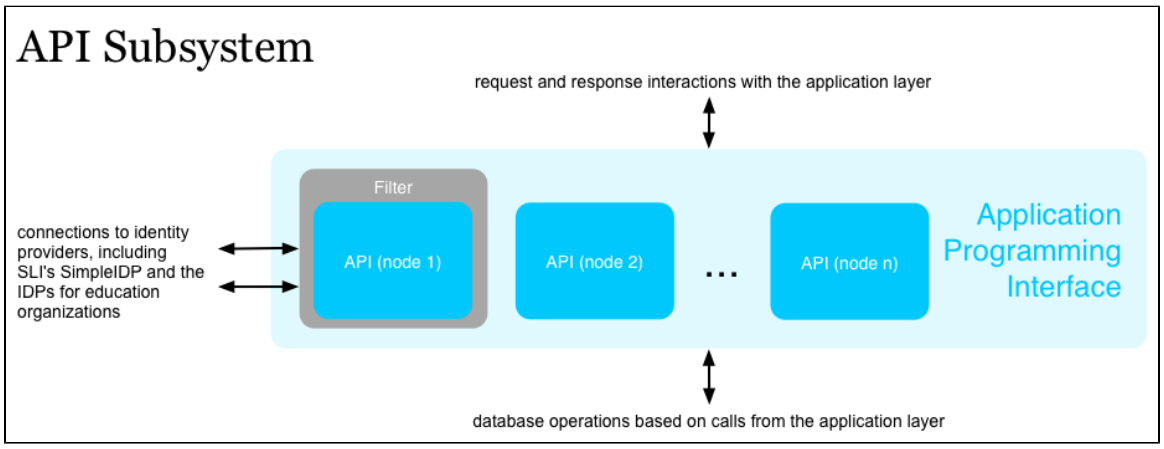
At the end of this process, all of the entities' references have been replaced by deterministic ids.

SLI API

Subsystem Overview

SLI uses the Representational State Transfer (REST) architectural approach to its application programming interface (API). SLI provides API as a series of stateless Java nodes that respond to HTTPS requests and provide authenticated, authorized access to data from the SLI data store. An API filter inspects each HTTPS request for a security token and redirects unauthenticated requests to an identity provider.

The following diagram and tables introduce the components and connections in the API subsystem.



Components

Component	Description	SLI Role
API node	Java web application	Provides secure, RESTful services for SLI data store interaction
Filter	A logical layer inside of the API nodes	Validates that an HTTPS request is authenticated and, if not, redirects the call to an identity provider

Connections

Connection	Description
1	Application layer traffic (HTTPS)
2	Authentication and context resolution with identity providers (SAML over HTTPS)
3	Database interaction for CRUD operations on data (Mongo Wire Protocol over TCP/IP)

Supporting Components

The API nodes rely on these supporting services:

- Tomcat web application server
- The *mongos* routing service, routing database traffic to the appropriate MongoDB node in the data store subsystem
- Common Linux operating system processes

REST and HATEOAS

SLI relies on REST protocols for sending and receiving API calls. SLI also makes use of HATEOAS links for navigation between entities.

As a result of this RESTful approach, the API is independent of the code used to make the API calls. Applications built for the SLI platform may exist in any language that supports making HTTP calls. This includes, but is not limited to, Java, .NET, JavaScript, Ruby, Python, C++, and PHP. Currently the default applications in SLI, using the API, are written in Java and Ruby.

Service Layer

The service layer enables each resource to gather entity specific information from the datastore. It:

- Converts entities from the their datastore representations to the exposed representations
- Exposes CRUD methods that act on entities
- Injects teacher contextual resolution security
- Applies entity attribute level access policies

Entity/Association Service

Each entity definition encapsulates its own service instance. The service instance is populated at the time of the creation of the entity definition. This enables each entity type to easily call its service methods without any lookups. There are two main service types for each entity type in the system:

- Entity Service - Handles service methods for main entities
- Association Service - Handles service methods for association entities

Entity Definition

An entity definition allows the API to recognize a serviceable resource. This could be a student, sections, student section association, etc. Each entity definition contains the following objects:

- entity's type - each entity will be persisted with the same entity type
- entity's (persisted) collection name - each entity will be persisted to a collection with this name
- entity's schema, including easy access to reference fields - used for input validation and reference resolution
- access to the entity's repository - for CRUD purposes
- access to other entity definitions that refer to the current entity - for performing queries and providing relevant links to the end user

In addition to simple entities (student/section/etc) there are associations that relate two entities (ie student section association). Associations extend basic entity definitions and include both associated entities' entity definitions, for use in validating that the associated entities exist. This allows the API to serve both basic entities and association entities using the same functionality.

When posting any entity, the API will use the associated schema to validate the input. The API checks that all required fields are present, and that formats are correct (no text in numerical fields, etc). The API also checks all supplied references to make sure that the referenced entities exist. If any of these entity-level validations fail, the API will reject the create or update (POST/PUT) request.

Entity Treatments

Treatments are used to transform entities between database representations and exposed representations. This is an extensible framework where treatments can be added by conforming to an interface.

Entity treatments affect the data that is presented or stored by the API compared to what is persisted. This allows the API to automatically perform operations on all data that is returned by the API. These treatments affect both input and output of the API. In general, treatments are written to perform on both the input and output operations, to enable it to reverse any treatment(s) it applies. This means that any modification they do on output, they can reverse when input is processed. This allows full round-tripping of data between the API and the end user, even though the persistence of that entity is slightly different.

There are currently 3 treatments applied by the API that support both input and output operations:

Treatment	API Output Operation	API Input Operation
ID	Puts the entity's ID into the field "id"	Removes the "id" field from the input (if present), then attempts to persist the data
Metadata	Puts the entity's metadata into the field "metaData" for roles, custom roles, application authorizations, applications, and onboarding resources. All other types of entities do not include metadata	Removes the "metaData" field from the input (if present), then attempts to persist the data
Type	Puts the entity's type into the field "entityType"	Removes the "entityType" field from the input (if present), then attempts to persist the data

The API's entity treatments allow it to serve additional data to the user compared to what is normally stored for an entity. The same treatments allow the "treated" input to be modified, returned to the API, and accepted.

Data Access Layer

The Data Access Layer handles interactions with the Mongo database, via the MongoTemplate. It provided access to entity collections and sub-documents, queries, schema versioning, and data encryption.

PII Encryption

PII (Personally Identifiable Data) is encrypted when stored in the Mongo database. When an entity is stored in Mongo via the MongoTemplate, the EntityWriteConverter encrypts the PII data. The PII data is encrypted while the data is at rest in the database. When an entity is retrieved from Mongo via the MongoTemplate, the EntityReadConverter decrypts the PII data. The PII fields for relevant entities are specified in ComplexTypes.xsd.

- Encryption method used is AES/CBC with PKCS5Padding
- Initialization vector is provided via the sli.properties configuration file
- Secret key is stored in a JCEKS KeyStore

SliSchemaVersionValidator

SliSchemaVersionValidator provides a mechanism to update the schema of the collections in Mongo when new versions of the software are installed.

Entities in the SLI data model can be assigned a version number. This is accomplished by annotating the related complexType definition in ComplexTypes.xsd. When either the Ingestion service or the API service is started the version of the schema in the software is compared with version of the schema in the database for each entity collection. If a database collection is found to have an older version, the related entity collection in sli.metaData is flagged to be updated by a SARJE job.

MongoTrackingAspect

MongoTrackingAspect is a component which tracks calls to the Mongo template and Mongo driver. It collects performance information and logs queries that exceed a specified elapsed time.

MongoEntityRepository

MongoEntityRepository exposes methods for interacting with Mongo, including basic CRUD operations.

SubDocAccessor

SubDocAccessor is a utility for accessing subdocuments that have been collapsed into a super-doc. If an entity is determined to be a sub-doc entity, the MongoEntityRepository uses the SubDocAccessor to create a query that targets a sub-doc, rather than a top level collection. At a higher level in the code, the sub-doc entities are treated as if they were top-level independent entities. The SubDocAccessor handles database queries to support that abstraction.

Denormalizer

Some fields have been denormalized and exist in multiple locations in the database. When a update query is made that would affect one of those fields, the Denormalizer creates a query which will update the data in the additional locations as well. This keeps the redundant data in sync and accurate.

EntityKeyEncoder

EntityKeyEncoder encodes entity keys that would cause problems in Mongo. In some cases the performance tracker attempts to insert a field with a '.' in it, which mongo-java-driver-2.9.1 does not allow. The encoder replaces invalid field names with valid ones.

MongoEntity

When an entity is being created, MongoEntity generates and assigns the entity's deterministic id before saving to Mongo.

Custom Data

A feature of the API is the ability to store custom data attached to an entity, but not part of the standard API. This makes it possible to expand the functionality of your application without the need of a separate database or data store. For example, a user can save user preferences for an application as custom data attached to the Staff entity, or store extra information about an assessment as custom data attached to the Assessment entity.

Custom data is a sub-document, less than 16MB in size, which an application can use to append application level information to the core entity model. The sub-document is created, read, updated and deleted at the parent entity level and are exposed via CRUD operations on the parent level entity . For example, in an application:

```
POST /students/{studentId}/custom

GET /students/{studentId}/custom

PUT /students/{studentId}/custom

DELETE /students/{studentId}/custom
```

are used to manage custom data on students.

At this time, custom data is secure at the application level, which means one application can not access custom data created by another application. This is achieved by using an OAuth token as the ClientID of the application, and storing the ClientID as a key-value pair in the sub-document. Additionally, a user of an application can only have access to the application's custom data if the user has context to the core entity.

Validation

Validation of URL and validation of entity against the sli schema. The validation logic has two components, URL and Entity validation.

URL Validation

The validation layer that handles the URL has the following strategies in place:

Functionality

1. Simple URL Validator that uses the apache-common url validator with valid schemes of "http" and "https"

2. Query string validator that ensures the validation against the black listed characters or strings. Black listed strings includes null, null bytes, form feed, line feed, escape, carriage return etc

Entity Validation

This validates the entity during create/update (POST and PUT operations) against SLI defined schema and uses the following logic:

1. Ensures all mandatory fields are present in the entity body

2. Validates the entity using the natural keys to ensure the uniqueness of the entity. The natural keys can be identified using the following tag in the SLI schema:

```
<xs:appinfo> <sli:naturalKey>true</sli:naturalKey></xs:appinfo>
```

Response

The invalid requests can have following response codes:

```
409 : Natural key validation failed (entity is not unique) 400 : All bad request other than natural key validation
```

The response looks like:

```
{ "type":"Bad Request", "message":"Validation failed: ValidationError \[type=INVALID_VALUE,
fieldName={field}, fieldValue={request url}, expectedTypes=\[\]\]", "code": {request code}}
```

Query Filter and Search

Fast, robust functionality to search for students is available through the SLC Datastore API through specific search URIs.

The search is a case-insensitive, keyword search that finds "Starts with" matches on search terms. For example, the search term "mat" wouldl match student "Matt Sollars".

The API URIs supported are:

URL	Example	version
[https://api.slcedu.org/rest/v1/search]	[https://api.slcedu.org/rest/v1/search?q=mat%20so]	1.0
[https://api.slcedu.org/rest/v1/search/students]	[https://api.slcedu.org/rest/v1/search/students?q=ma%20so]	1.0

The search fields are:

Entity	Searchable Fields	PII
student	name.firstName, name.middleName, name.lastName, otherName.firstName, name.lastSurname	Y

Security

As with the rest of the API, the results returned for a user are limited by their security context - i.e. the school or education organization they are associated with, along with the sections they teach, etc.

Limitations

- Full substring search is not currently supported - i.e. typing "ollars" will not return "Matt Sollars" as a result
- No more than 250 results can be returned for any search

Architecture

Elasticsearch

Data to be searched is loaded from the main Mongo database into a separate Elasticsearch repository. This enables the SLC API to use Elasticsearch's robust built-in querying and clustering capabilities. Elasticsearch is a distributed, open-source search engine built on Apache Lucene.

Elasticsearch servers will run on separate servers from the API and the Mongo databases. Any PII (personally identifiable information) is encrypted at the file system level using Gazzang.

Search Indexer

A Java application called the Search Indexer loads data from Mongo to Elasticsearch. It can operate in two modes:

- Bulk indexing mode - performs a complete index of all the relevant data in Mongo
- Incremental indexing mode - continuously monitors Mongo for inserts, updates, and deletes, and sends the appropriate changes to Elasticsearch

API Versioning Strategy

Existing API URIs currently have the 'v1' token embedded in them. With this approach, both a major and minor version is included in the URI "v1.n". If the user only defines the major rev, he is internally routed to the latest major.minor revision. Versioning via a URI component is the recommended option since it is the common industry practice. However, it requires the introduction of new resources when updating the version. Developers must always explicitly call a specific major version and that URI will be supported until the major version has been disabled after a period of deprecation.

Updating individual resources with this scheme is difficult without breaking the assumptions made by consuming applications. Versioning via URI makes versioning individual resources difficult. If we update Student to 'v2', we might also update all other resources to 'v2' even if the resource has not changed. This simplifies things from the developer standpoint.

We also start to leverage the concept of "soft links" in the URI where V1 is a soft link to the latest minor release in the major release.

Version in URI	Notes
v1	Requests with just the major revision in header defaults to the latest major + minor version of the API
v1.n	Requests with explicit minor version headers are handled by the specified version of the API (v1.1, v1.2, etc.)

The following table explains what would cause API to get a major revision over a minor revision. Major versions are not be backwards compatible.

Type of Change	Minor version	Major Version
New Endpoints	Yes	-
New Fields	Yes	-
New media types	Yes	
Removing Endpoints	-	Yes
Removing/Restructuring Fields	-	Yes
Query Parameter Change	-	Yes

Resource Deprecation

Every effort will be taken to avoid deprecating API resources. Deprecation will not affect existing functionality. In the event where deprecation is deemed necessary, the follow process is required:

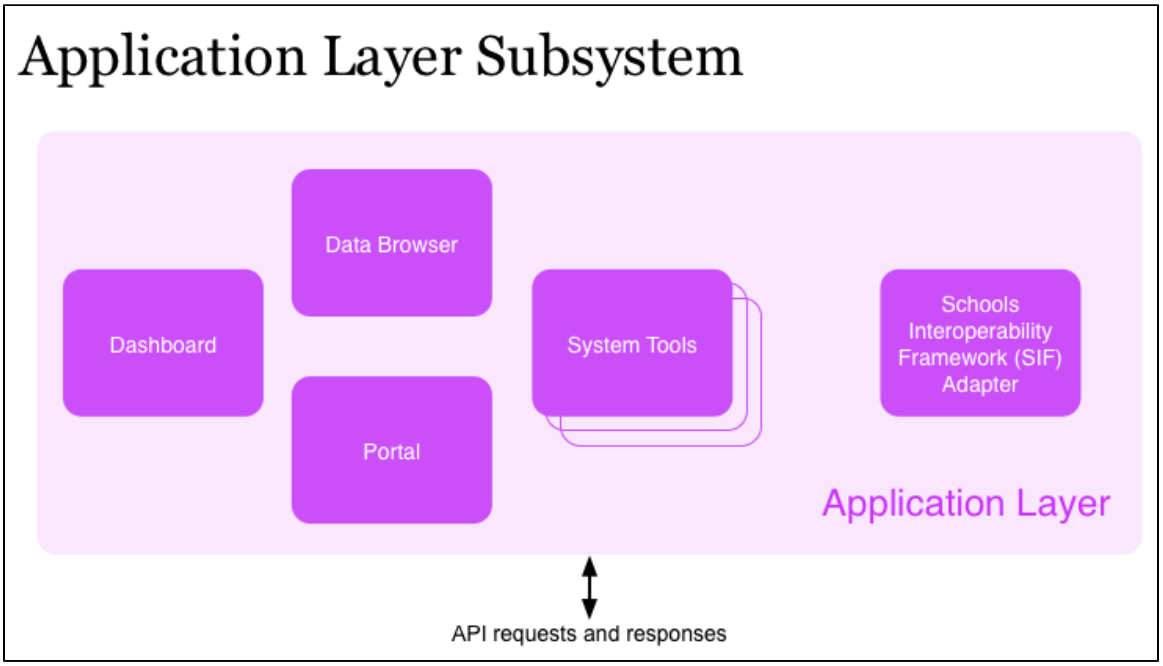
1. Discuss the proposed deprecation with the SLC and get approval.
2. Update the documentation to add the 'Deprecated' keyword. This keyword should indicate which version of the API deprecated the functionality.
3. Describe why the functionality was deprecated.
4. If an alternative approach is available, document this as well.
5. Deprecated features will be continue to function through a deprecation period (suggestion 12-18 mths) but no new functionality will be added to the deprecated feature.
6. During the deprecation period encourage the users to move to the updated functionality.
7. No published SLI RESTful API resource will be removed. The goal is to support client applications without requiring developers to update calls after a new version is introduced. The hope is to minimize impact on developers by having both alternatives and a deprecation period. There is also a goal that we do not attempt to support more than 2 major version at a time.

Application Layer

Subsystem Overview

The application layer is a group of web applications, each presenting a user interface. Each application has its own target users, such as system tools for SLI administrators and a customizable dashboard application for educators. Default applications in SLI include those shown in the Components table below. Also in this layer is the Schools Interoperability Framework (SIF) adapter for other applications developed to work on the SLI platform. Open source versions of the default web applications offer developers examples to work from.

The following diagram and tables introduce the components and connections in the application layer.



Components

The application layer is comprised of several user facing web applications. These Ruby on Rails applications provide both administration and configuration functionality, including administrative system tools and the data browser. The portal is a Liferay application that can contain other web applications to provide a consistent look and feel to the user.

Component	Description	SLI Role
Dashboard	Java web application	Educator application for creating and using dashboards that display student data
Data Browser	Ruby on Rails web application	Administrator tool for viewing contents of the data store, typically for verifying or validating a successful data ingestion operation
Portal	Java-based Liferay portal application	User portal providing a unified view of web applications available for the current user in SLI
System Tools	Ruby on Rails web applications	An Admin page and a series of web applications linking from that page for making administrative API calls
SIF Adapter	SIF adapter	Java application that forms a data integration bridge between SLI's API nodes and a SIF message pipeline, requiring a connection to existing SIF infrastructure for an education organization

Connections

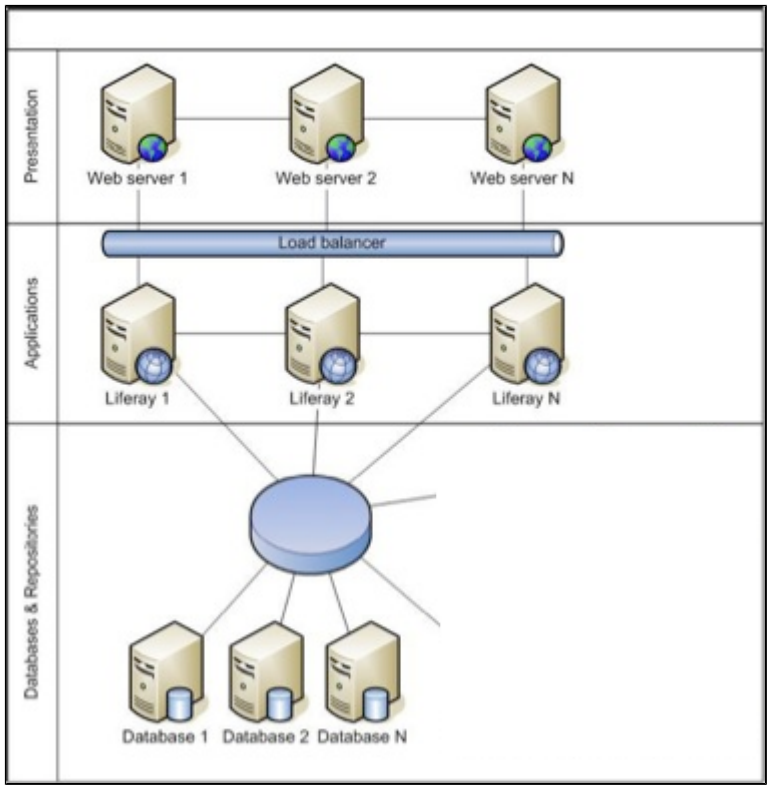
Connection	Description
All communication between applications and the API	HTTPS connections for requests and responses

End-user Applications

Portal

The SLI portal has developed on Liferay Portal Server and customized to meet SLI requirements:

- User Management, Portal Administration, Themes, Templates, Look & Feel Customization
- High Availability and scalable
- Open source project and supported by open source community
- Easy to intall and use and less expensive to maintain
- Working with various Operating System, Database, and Application servers



Production Portal

When a user is authorized by SLI single sign-on, the SLI production portal provides a home page on which there are a list of available applications. From this list, a user can click on a link that brings them to external SLI application. Also, the portal provides a header and a footer which the SLI application can pragmatically import into an application. The header provides a link to the home of the portal, a name of the user, a page to "Report-a-problem", and a link to log out. The footer provides a link to SLI privacy policy and term of use pages.

When an administrative users log in to the production portal, the portal provides an Admin link at the top of the page. This link takes the user to the Admin page that can be used to administer the portal instance.

Sandbox Portal

The sandbox portal is how SLC operators and application developers can perform administrative tasks related to a sandbox instance of the SLI. The separation of the sandbox portal and the sandbox allows developers to impersonate the user experience by logging in to the sandbox as a user while still being able to perform necessary administrative tasks in the sandbox portal.

Using an Alternate Portal

An alternate portal needs to handle:

- SLI Reset API
- SLI SSO
- Providing header and footer for applications
- OAuth
- Term of use for the first time log in

Using Alternate Header/Footer

SLI applications (such as SLI dashboard) rely on the header and footer from the SLI portal. Therefore, an alternate portal needs to provide an alternate header and footer.

Alternate Header

Can contain

- A link to the portal homepage.
- A page to report a problem to a system administrator.
- A link to logout (SLI Dashboard page does not provide a link to logout)
- Displaying full name of a current user.

Alternate Footer

Can contain

- Copyright information
- A link to SLI term of use
- A link to SLI privacy policy

Dashboard

The dashboard is a highly-configurable application used for the reporting and visualization of SLC data. It displays information about schools, districts, courses, sections, and students, along with student data such as attendance, assessment scores, and grades. The dashboard brings together this diverse set of data into holistic views, instead of requiring educators to search for data spread across many different applications. In this way, it highlights one of the most powerful advantages of the SLC platform - data all in one place.

The dashboard is a web application that SLC users can access using a standard web browser. It is also configurable by individual school districts (via the dashboard builder feature) to meet the district's particular data needs.

In addition to being a useful tool in itself, the dashboard is also a reference implementation for application development on the SLC platform. The code is open-source and freely available for download.

Features

Profile pages

These are the profile pages that currently exist:

- Educational Organization
- School
- Teacher
- Section
- Student

Panels

The dashboard comes with a library of predefined panels that may be added to the profile pages using the dashboard builder (see below). These include:

- Core state educational organization info
- Core district info
- Core school info
- Core teacher info
- Core section info
- Core student info
- Student assessment history

- Student current courses and grades
- Student attendance calendar
- Student transcript history
- List of students in section
- List of schools in an educational organization
- List of subjects and courses in a school
- List of teachers in a school
- List of sections for a teacher

Architecture

Drivers and Key Concepts

- Robustness
- Tested, reliable technologies
- Modularity, separation of layers
- Performance
- Ease of build and deploy
- Cross-platform compatibility

Technology

- Java
- Spring, Spring Modules
- Freemarker
- jQuery, jQueryUI, jqGrid
- Bootstrap
- Raphael
- Angular.js
- Maven

Testing Tools

- JUnit
- QUnit
- Cucumber
- Selenium
- JSTestDriver

Modules

The dashboard code base is organized into modules, or layers, with distinct responsibilities, that work together to retrieve data from the SLC data store and presents it visually to the user.

- Web Controllers - handle requests from the user's browser and send responses
- Layout Assembly - based on configuration files, assembles the data needed for all components of a page
- Managers - perform services (e.g. filtering, aggregation) on data, as well as caching
- Data Access - communicates with the Datastore API using the SLC SDK for Java
- User Interface - builds HTML and uses JavaScript for dynamic functionality in the user's browser

Dashboard Configuration Language

Dashboard is configurable through JSON driver config file. Types of configuration:

- Layout: Layout is a configurable UI component that has a notion of a header, navigation, and grouped content (panels).
- Panel: Panel is configurable UI component that consist of one or many widgets and represent a logical grouped unit of information or functionality, e.g. Contact Info, Grades, Enrollment History.
- Widget: Widget is a re-usable UI element with common population interface .e.g. drop-down, grid, tab menu.

Config Anatomy

```
{
  id : "template",      # unique identifier for the component
  type : "PANEL",       # PANEL, GRID, LAYOUT, WIDGET, TAB, FIELD
  data :{               # data associated with the component
    entity: "student",  # entity mapping
    cacheKey: "student",# data will be cached under this key in the model for the layout
    params: {           # Optional: params map to be passed to the entity retrieval method
      someint: 1,
      somebool: true,
      somestring: "xyz"
    }
  },
  params: {             # Optional: anything extra to be passed to the FE
    someint: 1,
    somebool: true,
    somestring: "xyz"
  },
  items : [             # Optional: view definition to be used with an existing widget, such as grid
(no other exist at this point)
    {id: "col1", name: "Name", type:"FIELD", datatype: "string", description: "Name of the Student",
field: "name.firstName", width: 80},
    {id: "col3",
     name: "Programs",
     condition: {       # Optional: condition to show/hide the item it's on, depending on the entity
field matching one of values.
       field: "limitedEnglishProficiency",
       value: ["yes"]},
     datatype: "programs",
     width: 80,
     formatter: Lozenge
    }
  ]
}
```

Layout

Constructing a Layout

```
{
  id : "student",
  name: "SLC - Student Profile",
  type: "LAYOUT",
  data :{
    entity: "student",
    alias: "student"
  },
  items: [
    {id : "csi", name: "Student Info", type: "PANEL"},
    {id: "tab6", name: "Overview",  type : "TAB", items: [{id : "enrollmentHist", type: "PANEL"}]},
    {id: "tab2", name: "Attendance and Discipline", type : "TAB", items: [{id : "csi", type:
"PANEL"}]},
    {id: "tab3", name: "Assessments",  type : "TAB", items: [{id : "csi", type: "PANEL"}, {id : "csi",
type: "PANEL"}]},
    {id: "tab4", name: "Grades and Credits",  type : "TAB", items: [{id : "csi", type: "PANEL"}]},
    {id: "tab5", name: "Advanced Academics",  type : "TAB", items: []},
    {id: "tab1", name: "ELL", condition: {field: "limitedEnglishProficiency", value: ["yes"]}, type :
"TAB", items: []}
  ]
}
```

Panels and Widgets

Constructing a Panel

The following example shows how to construct a new panel in dashboard.

```
{
  id : "contactInfo",
  type : "PANEL",
  name : "Contact Information",
  data :{
    entity: "student",
    cacheKey: "student"
  },
  params: {
       layout: ["student"]
  }
}
```

Constructing a Panel with a Grid

The following example shows how to construct a dashboard panel that contains a grid.

```
{
  id : "enrollmentHist",
  type : "GRID",
  name : "Enrollment History",
  data :{
    entity: "student",
    cacheKey: "student"
  },
  params: {
    layout: ["student"]
  },
  items : [
    {id: "col1", name: "School Year", type:"FIELD", datatype: "string", field:
"studentSchool.schoolYear", width: 100},
    {id: "col2", name: "School", type:"FIELD", datatype: "string", field:
"studentSchool.nameOfInstitution", width: 220},
    {id: "col3", name: "Gr.", type:"FIELD", datatype: "string", field:
"studentSchool.entryGradeLevel", width: 80},
    {id: "col4", name: "Entry Date", type:"FIELD", datatype: "string", field:
"studentSchool.entryDate", width: 120},
    {id: "col5", name: "School Choice Transfer?", type:"FIELD", datatype: "string", field:
"studentSchool.transfer", width: 180},
    {id: "col6", name: "Withdraw Date", type:"FIELD", datatype: "string", field:
"studentSchool.exitWithdrawDate", width: 120},
    {id: "col7", name: "Withdraw Type", type:"FIELD", datatype: "string", field:
"studentSchool.exitWithdrawType", width: 120}
  ]
}
```

Dashboard Builder

The dashboard builder is a feature of the dashboard that lets IT administrators configure data and functionality of the dashboard for their specific educational organization.

Dashboard page layouts are driven by JSON configuration, and the dashboard builder is a user-friendly interface for editing these configurations.

The dashboard comes with a library of defined panels, such as attendance data, school enrollment history, list of students in a section, and many others, which can be placed on pages by using the dashboard builder.

Architecture

The dashboard builder is part of the Dashboard web application, and uses the same basic technologies as the Dashboard: Java on the server-side, and JavaScript on the client-side. JSON configuration files are stored as custom data in the SLC data store, attached to the educational organization entity. Whenever changes are made in dashboard builder, AJAX calls are made to the dashboard server, which then updates the custom data through the SLC data store API.

The configurations follow a "waterfall" hierarchy, according to the educational organization hierarchy of the user. Default configurations are override by state-level configurations, which are in turn overridden by district-level configurations.

Data Browser

The data browser is a simple web application that allows administrators to verify that data exists in the SLC platform and to explore the API. This is written using Ruby On Rails and acts as a very thin interface to the API.

The application itself makes use of the HATEOS links that the API generates to present navigational choices to the user and represent the returned results as either sortable tables or as the explicit details of each individual result returned. It also allows for very simplistic searching based on things like staffUniqueStateId, studentUniqueStateId, stateOrganizationId, etc.

System Tools

The set of system tools available (Application Registration, Application Approval, Application Authorization, Realm Management, Custom Roles, etc) are all actually one application. This application does many different tasks, so each are separated in responsibility and security concerns. For example, someone who logs into the Admin app to manage realms, must be able to actually manage realms in order to access that page.

SLI Security

** THIS PORTION OF THE TECHNICAL SPECIFICATION CONTAINS DRAFT CONTENT THAT IS CURRENTLY UNDER REVIEW BY THE SECURITY OFFICE.

Internal

Server-level Security

Operator documentation, the "SLI Runbook," provides recommendations for configuring the various server software used in the infrastructure, such as Red Hat Enterprise Linux and Tomcat. These configurations DO NOT include the environmental hardening for secure operations that are necessary to ensure the security in the underlying platform. The overall configuration and hardening requirements may differ depending on the specifics of the operational environment in which the software is deployed. In all cases, operators should adhere to industry best-practices for the hardening of system components.

Furthermore, the security of PII that is transferred amongst the many server components and stored within the Disk I/O subsystems of the same server components is not provided by the SLI software. In all cases, additional 3rd party open-source or commercial components will be required to provide operational security for PII transmitted by and within the system, or stored by the system. These requirements are solely the responsibility of the operator of the SLI instance.

Communication Between Servers

The network architecture between SLI components require only forward name resolution and IPv4 connectivity. The software had not been tested with IPv6, so IPv6 should be disabled, if possible, in a production deployment.

SLI does not require a direct connection between the SLI identity provider (SimpleIDP) and the identity providers of education organizations. However, the SAML assertion created by the education organization's IDP must be signed with a certificate that the SLI API can trust. This means the certificate is signed by a trusted Certificate Authority (CA).

Encryption of PII

PII (Personally Identifiable Data) is encrypted when stored in the Mongo database. When an entity is stored in Mongo via the MongoTemplate, the EntityWriteConverter encrypts the PII data. When an entity is retrieved from Mongo via the MongoTemplate, the EntityReadConverter decrypts the PII data. The PII fields for relevant entities are specified in ComplexTypes.xsd. The PII data is encrypted while the data is at rest in the database.

- The encryption method used is AES/CBC with PKCS5Padding.
- The initialization vector is provided via the sli.properties configuration file.
- The secret key is stored in a JCEKS KeyStore.

Blacklist Enforcement

The SLI system accepts input from many sources, including the API, the ingestion engine, and the on-boarding process. Each source of input allows the possibility of inserting malicious code, such as cross-site scripting (XSS). To prevent this, it was necessary to implement one shared set of rules that can be performed on this input. These rules reject clearly malicious code and block it from entering the data store.

Implementation

A blacklist validation mechanism is implemented inside the domain portion of the data access layer. Data saved to the data store must pass through this validation. Since this is not enough to protect a user of the data from malicious data, any code that uses data from the data store as input (which includes browsers) should implement a strategy of sanitizing and encoding for true protection.

A framework has been added to define various strategies for validating input. Initially, three strategies are defined:

- **Blacklist by characters (`CharacterBlacklistStrategy`)** -- Given a set of blacklisted characters, ensures that the provided data does not contain any of the characters. The set of characters is configurable.
- **Blacklist by words (`StringBlacklistStrategy`)** -- Given a set of blacklisted words, ensures that the provided data does not contain any of the words. The set of words is configurable.

- **Blacklist by regular expressions (`RegexBlacklistStrategy`)** -- Given a set of regular expressions, ensures that the provided data does not match any of the regular expression patterns. These validation rules are more expensive, so should be used only when `StringBlacklistStrategy` is insufficient. The set of regular expressions is configurable.

Levels of blacklist validation

Two levels of blacklist validation are implemented:

- **Strict** -- Used by default on all strings, validates against largest sets of characters, words, and/or regular expressions.
- **Relaxed** -- Explicitly configured in the SLI schema for a few exception fields that may contain data blocked by the strict validation rules. To further protect the relaxed fields, the 'blacklist by words' strategy is also used to look for words that, in combination with the '<' bracket, can cause problems.

Administration

Realm and Tenant

A realm administrator can use the Realm Management system tool to create, delete, or modify a realm. Each realm is associated with an education organization, and since each realm administrator is tied to a single ed-org, the realm administrator can only administer a single realm.

The realms appear on the realm selection page when a user tries to authenticate. Each realm contains:

- Display Name - Name of the realm that appears on the realm selection page
- IDP URL - Identifier for the IDP
- Redirect Endpoint - URL of the IDP where the user will be directed for authentication
- Realm Identifier - Unique value that identifies the relam and can be used by an application to bypass realm selection

A tenant is roughly equivalent to a state in our system. Users have the ability to see data of other users within the same tenancy (if authorized). It is not possible for a user in tenant A to see data in tenant B.

State and Local Education Agencies

SEA and LEA administrative user roles are tied to state or local education agencies. These roles control access to data based on tenancies and realms and define the permissions of the user. SEA Administrators administer state-level instances of the SLI platform. This includes creating new administrative users and enabling applications for delegated child education organizations. LEA Administrators administer district-level SLI instances. Typically, this includes creating new administrative users, and enabling and configuring applications for a district.

Delegation

An LEA Administrator has the ability to authorize applications for use within the local education agency (LEA). However, the LEA Administrator can choose to delegate that responsibility to an SEA Administrator using the Admin Delegation system tool.

Once delegated, the SEA Administrator has the ability to use the Application Authorization system tool to authorize and deauthorize applications for the LEA. The LEA Administrator cannot use the Application Authorization system tool as long as that authority is delegated, but can change the delegation at any time in order to regain authorization privileges.

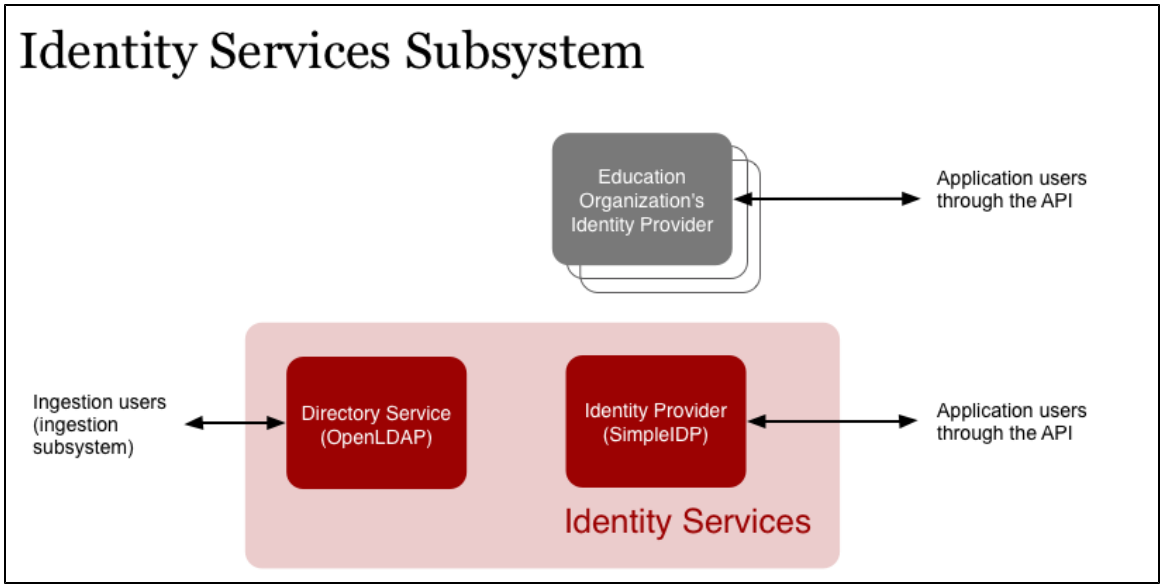
Application and Data Access

Integrated Identity Services

SLI includes an identity service called SimpleIDP. SimpleIDP and the SLI's directory service (using OpenLDAP) handle authentication for SLI Operators. For SLI administrators and end-users, SimpleIDP integrates with an education agency's identity provider to help determine a user's permission to access applications, system tools, and data in the SLI.

Applications rely on identity services for logging in and for obtaining security tokens for asynchronous database transactions. Identity services use Security Assertion Markup Language, version 2 (SAML2) over HTTPS. Applications obtain and use OAuth2 authorization tokens through the API.

The following diagram and table introduce the components in the ingestion subsystem.



Component	Description	SLI Role
SimpleIDP	Identity Provider	Java web application that acts as a simple SAML2 gateway for authenticating operators and administrative users
OpenLDAP	Directory Service	Directory service supporting SimpleIDP that contains accounts for SLI operators and authenticates users of the ingestion FTP service
State and District IDPs	Identity Providers	Authentication services for an education organization's users, whose information is in the SLI data store and who have access to data in the data store. Referred to as *federated users*, these users have a unified single-sign-on experience when using applications in the SLI.
Search	*(Not Yet Implemented)*	*A service for selected student data search*

Authentication Flow

Before looking at the flow, let's start with the list of connections shown in the Identity Services Subsystem diagram:

- SAML2 traffic between the API and the SLI's identity provider for authorization (API subsystem)
- SAML2 traffic between the API and an education organization's identity provider for authorization (API subsystem)
- SimpleIDP interaction with SLI's directory service
- Directory service authorization request for ingestion users (ingestion subsystem) using PAM (Pluggable Authentication Module)

Here's a summary of the user authentication procedure using this federated approach:

1. When signing in to an application or service in the SLI, a user is prompted for an email address and password. The user is expected to enter the credentials associated with the education organization's directory or SSO service. In other words, the user provides the same email address and password used when signing in to other applications used by that education organization. SLI makes this request using SAML 2.0.
2. The identity provider for that education organization authenticates the user and returns a SAML 2.0 assertion to the SLI service provider indicating that the user provided correct credentials. This SAML assertion includes an attribute with role and UID data from the identity provider.
3. The SLI authorizes the authenticated user to access applications and data based on how that user's roles in the education organization have been mapped to roles in the SLI. This role mapping is required for users to access any applications or data in the SLI. See the User Access Control security sections on user roles and rights for more information about this mapping.

SAML 2.0

SAML is a protocol for requesting another server authenticate a user and provide information back about that user. SAML works for our API, operating as a SAML Service Provider, to communicate with multiple Identity Providers to authenticate the end user. It provides both authentication of the user and a means to control authorization based on the roles returned in the SAML Assertion. SAML does not solve the application authentication.

Simple-IDP is a Java Web Application that uses an LDAP server as its datastore. It operates as a SAML Identity Provider responding to SAML requests with SAML Assertions. It provides authentication means for non-federated users; i.e., administrators and application developers.

Simple-IDP was implemented to avoid hosting users within the SLC API, to make use of the API's ability to authenticate users via SAML, and because most existing SAML IDPs are large, complex systems that provided more features and overhead than what the SLC needs. The Simple-IDP is simple. It responds to a SAML request, displays a log in page, authenticates a user against an LDAP server, then returns a SAML Assertion. Configuration of the Simple-IDP and API is also very simple, with trust information stored in properties files on each server.

Production Mode

In production, the Simple-IDP authenticates all non-federated users. This includes the SLC Operators, SEA Admins, LEA Admins, and Application Developers. This gives users access to the administration portion of the SLC system. Those user accounts are created via the Admin Tool in LDAP. The Simple-IDP has no datastore of it's own.

Sandbox Mode

In sandbox mode, the Simple-IDP authenticates application developers and SLC Operators. This gives users access to the administrative portion of the SLC system.

The Simple-IDP also supports impersonation when in sandbox mode, which allows a developer to simulate logging in as a federated user. The two different modes (administrative and impersonation) in sandbox are supported via configuration of two different authentication realms in the API. Sandbox users can choose whether to administer their account or test their application as a federated user. The Simple-IDP therefore operates as two different IDPs in Sandbox Mode. During impersonation the application developer logs into his developer account (which authenticates him against his LDAP credentials) then selects the user to impersonate and a SAML Assertion is created and returned to the API allowing him to impersonate whatever federated user he wants to.

OAuth 2.0

Authentication on the SLC Datastore is performed via OAuth 2.0. Through the OAuth process, the application receives an authentication token representing an SLC user. This token must be saved and used whenever a call to the Datastore API is made on behalf of the user.

The SLC API is a platform for application developers to write applications to access student and school data. This requires user authentication and authorization but it also requires application developer authentication and authorization. OAuth is a standard protocol with wide adoption for similar APIs. It provides a mechanism to authentication and authorize the application developer as well as the end user.

We combined OAuth with SAML to build an authentication and authorization framework that allowed for:

1. Authenticating the application (and application developer/vendor)
2. Authenticating the end user via their IDP, therefore not requiring new user credentials
3. Getting authorization information about the end user from their IDP via Roles/Group membership that can be used to drive their access to data in SLC
4. (future) Application authorization so that applications can be given restricted permissions

Federated Authentication Strategy

SLI relies on external Idenity Providers (IDP's) to supply authentication and role part of authorization. A user that wishes to use an SLI application is be re-directed to his/her IDP (typically hosted by SEA or LEA), to log in. The IDP then redirects the user back to SLI with a "vouching message", a SAML2.0 Assertion, that describes who that user is and what he is allowed to do.

This setup allows for States/Districts to re-use their existing identity stores.Users only enter their credentials on the site they trust, never revealing those credentials to SLI, and SLI is liberated from managing a user's authentication-related activities (log in, password reset, password expiration, multi-factor authentication...etc).

Session Checks

A user's session with one SLI-enabled application can transparently propagate to another SLI-enabled application. For example, if user logs in to use the data browser and then uses the dashboard, the user is not required to re-authenticate. The session will propagate transparently, providing a seamless user experience.

Note: Sessions can only propagate between SLI-enabled applications.

User Access Control

Account Management

The Admin Account Management Tool (AAMT) and Developer Account Management Tool (DAMT) perform similar roles, but in different environments. DAMT is used to create and manage developer accounts within the sandbox environment. AAMT allows SEA or LEA Admins to create additional admin users within their state or local education agency. These users may be assigned one or more roles that define their

permissions.

For both AAMT and DAMT, user accounts are created within SLI's hosted LDAP directory, and authentication is provided by SLI's IDP. Both tools are Ruby on Rails applications that call out to RESTful APIs.

Rights

A right is the smallest unit of an access grant. Fields within data are tagged with a list of rights which enable a user to see (or not see) that field. Roles are composed of rights. User's roles are decomposed into a list of rights. That list is consulted while the data is being retrieved or written to make a decision if some data should be filtered out, allowed to go through, or flat-out denied.

Roles and Permissions

Different SLC users have different roles, such as staff, teacher, IT Administrator, etc. Each user has his own security context that is based on the role and the school or educational organization he is associated with, which determines the user's permission to access a specific subset of data.

For example, a teacher is only be able to see the students belonging to the sections that she teachers.

Custom Roles

The Custom Roles system tool allows a realm administrator to control how rights are assigned to users who authenticate against a realm's IDP.

When a user authenticates against an IDP, the IDP includes a list of roles assigned to the authenticated user as part of the SAML response. Since the roles vary from IDP to IDP, it's necessary to specify which SLC-specific rights are assigned to those roles.

The realm administrator can define one or more role groups, each of which contains a

- Group name - descriptive name for the group
- Role name - the names of one or more roles to match against the roles from the IDP
- Rights - the SLC rights which will be assigned to the role(s)

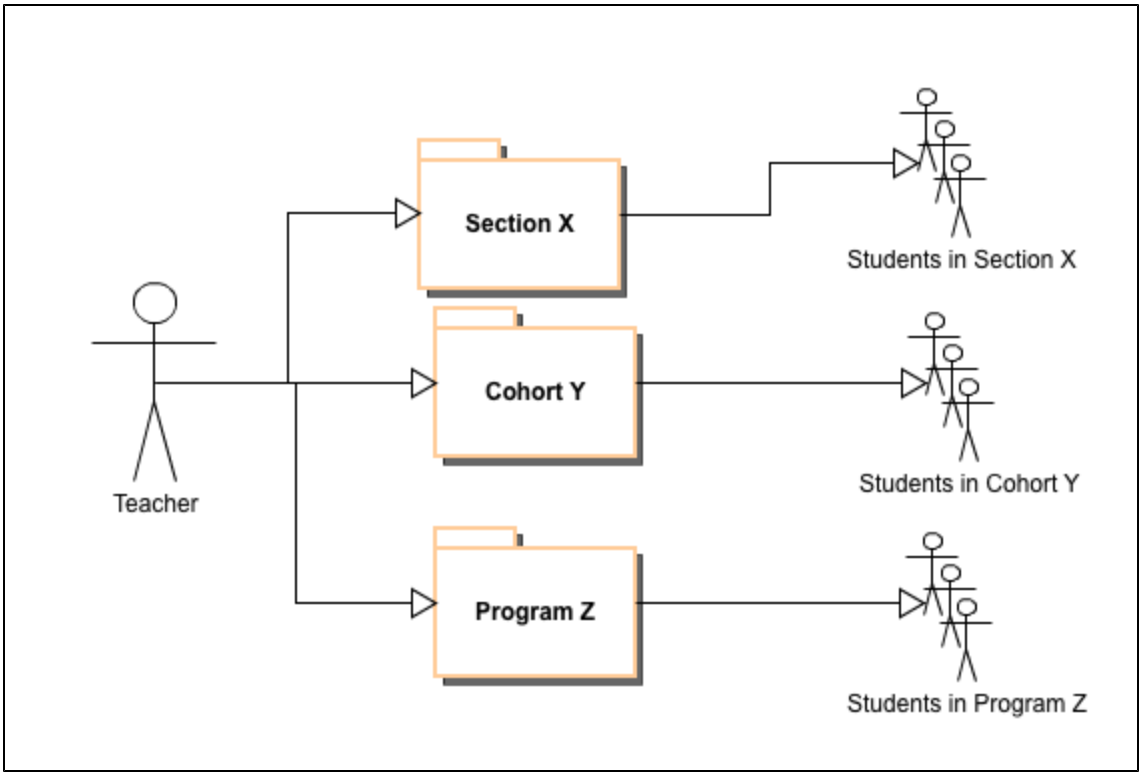
When a user logs in, the roles from the SAML response are matched against the role names in each role group, and the user is granted the rights from each matching group.

Context Resolution

The API enforces contextual authorization rules that help determine a user's access to data.

Teacher Contextual Authorization

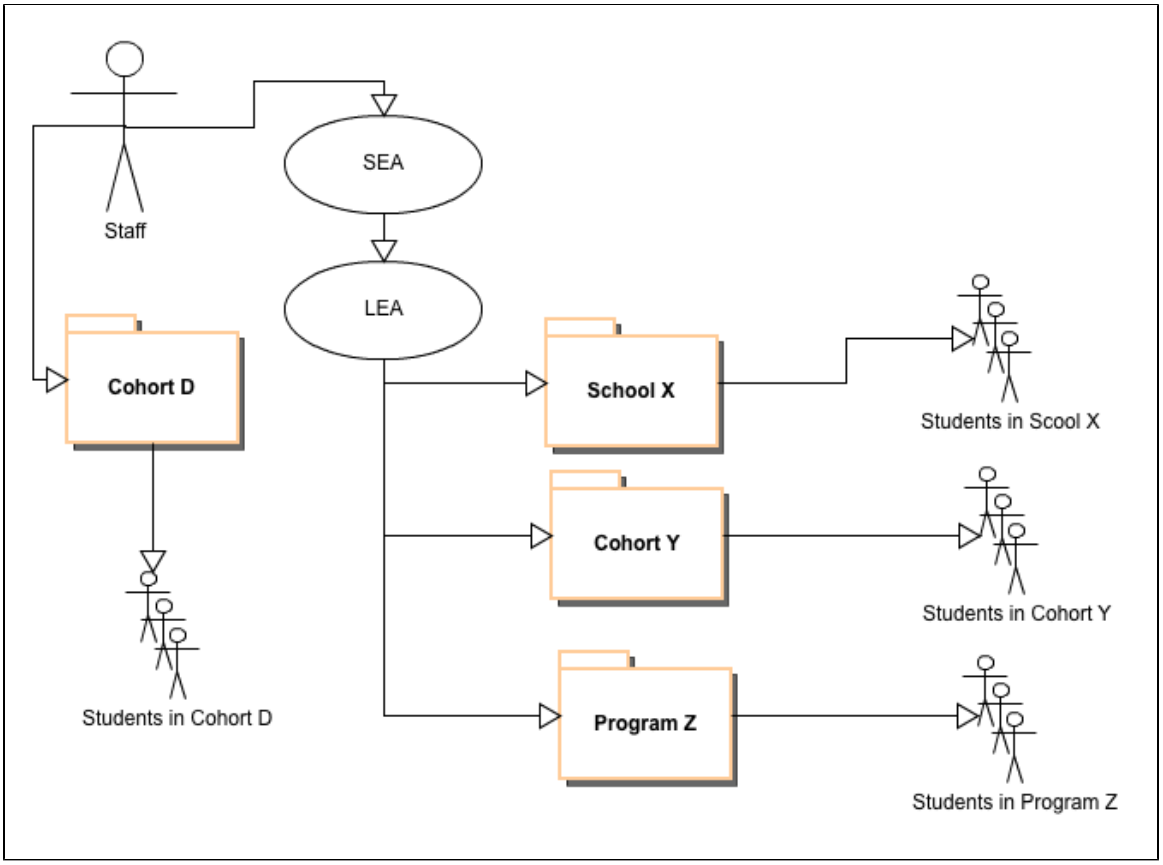
Teachers gain access to students (and student data) through being assigned to sections, programs or cohorts.



Staff Contextual Authorization

Staff can see resources in the SLI to which they are connected to via Education Organization Hierarchy. This means all students (and student data) which attend schools in their district(s). This also means all students (and student data) which are assigned to programs/cohorts in their EdOrg hierarchy.

Staff can also see students (and student data) in cohorts/programs to which they are assigned and can see other staff/teachers in their EdOrg hierarchy.


Staff
SEA
LEA
Cohort D
School X
Cohort Y
Program Z
Students in Cohort D
Students in Scool X
Students in Cohort Y
Students in Program Z

Back-End Operations

Redundancy

To protect against server failure, each server or component should have some type of secondary server or running component. MongoDB uses replica sets with three members so that all data is stored three times and MongoDB's failover mechanism can handle server failures. For other types of servers, redundancy is achieved by having at least one secondary server and leveraging a load balancer to minimize end user impact.

Scaling

Scaling of front-end servers (API, Dashboard, Portal) is done by adding additional application instances behind the load balancers. Details for each specific front-end application is covered in each application's section of the platform runbook. Scaling of database servers is accomplished by adding shards, where each shard is a MongoDB replica set of three servers.

Sizing

This is a summary of the recommended hardware for environment sizing. The Environment Specification document provides configuration details for each type of server. Estimates for data store infrastructure are based on use of Amazon m2.4xlarge size instances, or equivalent hardware. Use of differing hardware specifications will naturally result in different shard counts for the data store.

Server Type	Minimal (with redundancy)	5 million students 4 years of data	Additional 1 million students
Portal	2 Tomcat 2 MySql	3 Tomcat 2 MySql	0.25 Tomcat
Dashboard	2 Tomcat	3 Tomcat	0.25 Tomcat
Data Browser	2 Ruby/Rails	2 Ruby/Rails	N/A
Admin	2 Ruby/Rails	2 Ruby/Rails	N/A
API	2 Tomcat	7 Tomcat	0.5 Tomcat
Data Store	3 MongoDB shards (9 servers) 3 Mongo config	23 MongoDB shards (69 servers) 3 Mongo config	4-5 MongoDB shards (12-15 servers)
IDP	2 SimpleIDP 2 OpenLDAP	2 SimpleIDP 2 OpenLDAP	N/A
Ingestion	2 Landing Zones 2 GlusterFS 2 ActiveMQ 2 Ingestion DeltaHash DB: 1 MongoDB shard (3 servers) 3 Mongo config Staging DB: 1 MongoDB shard (3 servers) 3 Mongo config	2 Landing Zones 2 GlusterFS 2 ActiveMQ 7 Ingestion DeltaHash DB: 3 MongoDB shards (9 servers) 3 Mongo config Staging DB: 1 MongoDB shard (3 servers) 3 Mongo config	DeltaHash DB: 0.5 MongoDB shards 1 Ingestion

Scaling and Sizing Variables

Number of Students and Quantity of Data per Student

Supporting more students, or more data per student, requires scaling the SLI database to hold more data. The SLI database is sized to allow holding all indexes and 10% of data in memory, so that most database queries are not disk I/O bound. If data is ingested via EdFi, scaling the DeltaHash database is also required to handle deduplication of a larger number of incoming EdFi entities. The DeltaHash database is sized to allow holding all data in memory to allow rapid deduplication of ingested EdFi entities.

More Years of Historical Data

Supporting more years of historical data requires scaling the SLI database as described above.

End User Driven Load

Supporting more API, Dashboard, or Portal load requires adding servers of the affected application and could require adding more SLI database shards. This is primarily expected to supported by scaling the API layer and with that the number of SLI database shards is expected to be driven by memory requirements more than API load. Therefore, more SLI database shards may not be needed.

Staging and Testing Updates

It is recommended that any operator of the platform make use of a minimally scaled down, yet redundant, test environment in compliance with minimums of the operator's own requirements. This is to serve as a testing ground for platform deployments, configuration validation, and upgrade testing.

Monitoring

While the platform does not require use of a monitoring and alerting system, it is recommended that any organization operating a deployment of a platform make use of such a system in order to identify potential problems before they are issues, or to help isolate root causes once a failure has occurred and been reported to the operator.

Additionally, it is recommended that any system used be capable of recording and displaying basic usage information over time in order to permit an operator to make informed capacity management decisions.

Load Balancing

An important component of any platform deployment is the ability to balance loads across multiple servers. While no specific load balancing method is recommended, the software has been thoroughly tested with Amazon Elastic Load Balancers, as well as the open source load balancing utility HAProxy, which can be found at http://haproxy.1wt.eu/.

The platform components have no expectation of session stickiness, although some level of session stickiness can help improve performance. This is because caching would not need to occur on multiple servers, so users would not to notice any momentary navigation stall as a result of data being loaded to additional servers (beyond the first server accessed).

However, the API should generally not be serviced with session stickiness. Session unique data is housed inside the DAL, and session stickiness to single API servers reduces the overall API scalability- as any API node is able to service any authenticated request.

Secure Sockets Layer/Transport Layer Security with Load Balancing

The platform was explicitly written expecting Secure Sockets Layer (SSL) /Transport Layer Security (TLS) to be in-place throughout the infrastructure of an operator's environment, behind any load balancers to each end server. This is due to the expectation that the infrastructure would be deployed entirely in cloud environments. It is recommended that any operator of the platform does not attempt to change this configuration.

Host Management

While the platform does not require use of a host management tool such as Puppet, Chef, or CFEngine, it is highly recommended that any organization working to deploy this platform do so by using such a tool in order to facilitate the creation and management of identical servers. The goal is to minimize differences and ensuring overall consistency.

Information on Puppet can be found at http://puppetlabs.com/.
Information on Chef can be found at http://www.opscode.com/chef/.
Information on CFEngine can be found at http://www.cfengine.com/.

Application Development on the SLI Platform

SLI provides developer resources for application developers to develop software to interact with the SLI data store.

Developer On-Boarding Process

Developer on-boarding is process by which a developer receives access to the SLI sandbox or production environment. Developers use this access to develop applications that work with the SLI data store and to learn the SLI platform.

Developer accounts are stored within SLI's hosted LDAP directory, and authentication is provided against SLI's custom Identity Provider (IDP).

Developer Sandbox Environment

There are two ways to create developer sandbox accounts:

1. First time registration for a sandbox account
   When a user registers for a sandbox account, a new Sandbox environment is created for them. Each Sandbox environment is isolated from all others - data is kept in a separate logical MongoDB database and configured applications are only accessible within that sandbox. The user with correct administrative rights can create multiple landing zones for the sandbox environment and use these to ingest data into their environments. Auto-loading of data is also available. When the Restful API in the sandbox environment responds to a request from an application, the API identifies which sandbox environment the user and application is associated with and routes the request to the correct logical database, ensuring no data outside the sandbox environment is accessible.
2. Developer is added to an existing sandbox account through an Developer Account Management system tool
   Users who are added through the the Developer Account Management system tool are created as members of an existing Sandbox environment. The creator may assign the new developer account admin rights or lesser rights, allowing the initial owner of the sandbox environment to restrict and manage access.

When logging into the sandbox environment, SLI's IDP presents the developer with the choice to log in as an administrative user or to impersonate an end user existing in the ingested data. The impersonation ability allows developers to test their applications without having to set up, federate, and maintain their own IDP for end user accounts.

Developer Accounts in Production

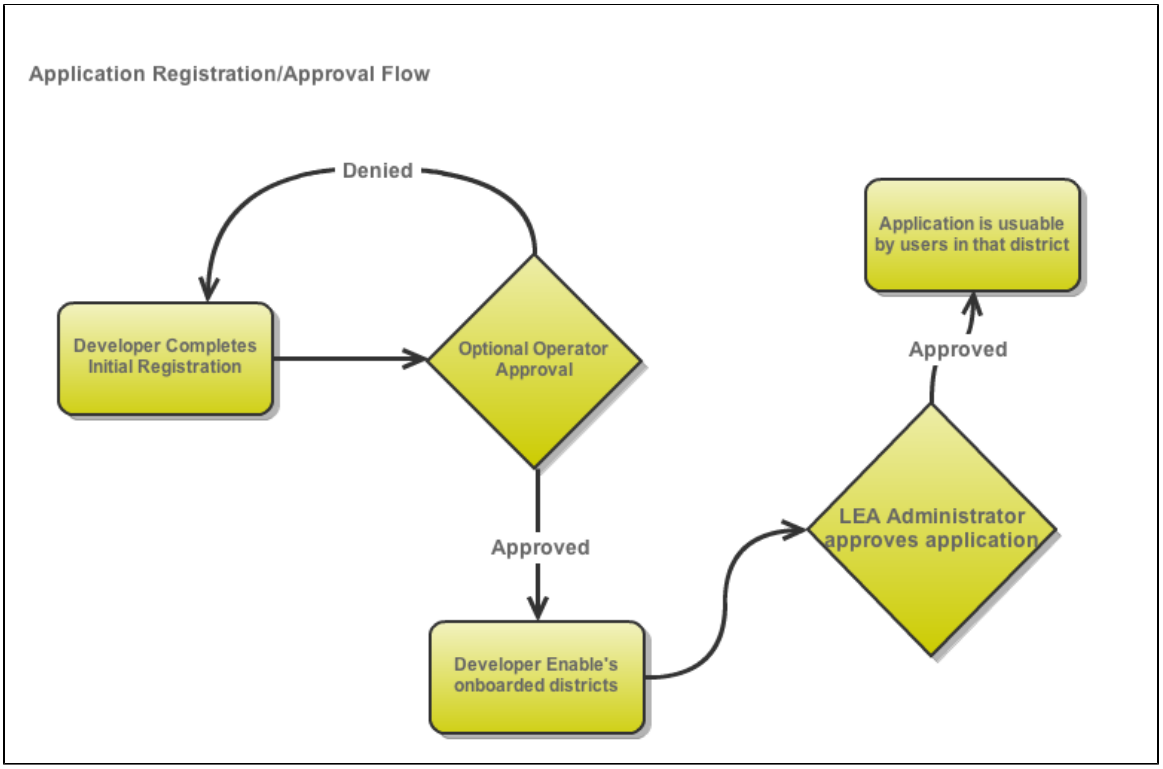
Developers do not have their own production instance, but are added to existing production environments. In production, the SLC Operator creates and manages developer accounts in order to ensure data security by limiting access to approved users only.

Applications Requirements

There is no requirement for an application to be developed using a specific language. Popular languages for application development include Java, .NET, JavaScript, Ruby, Python, C++, and PHP.

An application must be able to send HTTP requests to the SLI Data Store API. This will enable the application to authenticate and to send REST API requests. Using an OAuth library may help simplify the authentication process.

Application Registration and Enablement



Registration with the Desired SLI

After developers or vendors register for an account on either sandbox or production, they are able to register and manage their applications within the SLI. This functionality is provided through the administrative system tools.

When the developer registers an application, they must provide information that is required to facilitate OAuth, as well as to track changes to the application. This includes:

- Installed App checkbox- The Installed Application checkbox indicates to the SLC that this application uses a different OAuth workflow than the other applications. Checking the box disables the other options, but if this option is not used then the developer will need to provide an Application URL and a Redirect URL.
- Application URL- The Application URL is used to list the application in the portal application after it is approved.
- Redirect URL- The Redirect URL is used as a part of the OAuth flow.

After the application is registered and approved, the developer has access to a Client Id and Shared Secret, which allow the OAuth flow needed to access the SLI platform.

Enabled for One or More Tenancies in the SLI

After an application is optionally approved by the platform, a developer can enable the application to run for various districts. The registration form will contain a new option that allows a developer to select the districts within states that should be able to access to the application. The developer enabling the districts allows the districts to consider approval of the application to access their data.

Client ID and Secret Assigned

When an application is registered and approved, it is assigned a Client Id and Shared Secret. These must be saved and be accessible to the application when it runs. The application will supply the Client Id and Shared Secret when initiating the OAuth authorization process with the SLI Data Store.

OAuth Authentication

Authentication on the SLC Datastore is performed via OAuth 2.0. Through the OAuth process, an application receives an authentication token representing an SLC user. This token must be saved and used whenever a call to the Data Store API is made on behalf of the user.

Well-formed Calls to the API

The Data Store API provides access to a wide variety of data, such as educational organizations, schools, teachers, staff, students, attendance, and assessments. Data can be retrieved as singular entities, groups of entities, and groups of entities based around a single parent entity.

For example, to retrieve students, you can use:

1. /students/<id> to retrieve a single student
2. /students/<id1,id2,...> to retrieve multiple students
3. /sections/<id>/studentSectionAssociations/students to retrieve all the students in a particular section

The correct call to use depends on the functionality of the application and the flow of information, as well as efficiency, memory use, and code complexity.

Software Development Kits

The Data Store API has a REST interface, which allows for communication with any language or platform that can make REST calls. In order to facilitate the ease and speed of development, Client SDKs are provided for popular languages.

The SDKs make REST calls behind the scenes, but abstract the details behind an interface. A developer can make calls to get data from the SLC Data Store and the Client SDK will take care of the rest. Other benefits of the Client SDKs include: working with objects instead of JSON and JSON parsers, and consistency of use and error handling.

The client SDKs are available for download on GitHub.

Java SDK

Provides full functionality for connecting, authenticating, and sending requests to the API. The Java SDK is implemented as a light wrapper around the official reference implementation of Java API for Restful Services (JAX-RS). Requires Java version 6 or later.

Javascript SDK

Provides basic functionality for connecting and authenticating. Contains examples for sending requests to the API, which can be expanded upon. The JavaScript SDK uses the server-side JavaScript environment *Node.js.*

Releasing Applications to Production

After a developer or vendor has enabled registered applications for a district, the LEA Administrator can use the Application Approval system tool to approve or deny the application for the district. When an application is approved, users in that district will see the application as available for use, and the application will be able to access the district's data. If the application is denied, the application does not show up for users and the application will be denied to access any data in their system.

This means that even if a developer has an approved application that is enabled for that district, until the LEA administrator specifically approves it, users cannot use this application.

Tenant On-boarding Process

On-boarding is the process of setting up your Educational Organization to use the SLI. The on-boarding process involves setting up user accounts, ingesting data into the SLI, and [if applicable] enabling applications to access the data you have ingested.

Prerequisites

1. Must have two email addresses not already in use.
2. Must have a dataset that is ready to ingest.
3. Must have an SLC Operator account.

On-boarding with Bulk Ingestion

Educational Organizations on-board by setting up administrative accounts and bulk ingesting data into the SLI.

1. SLC Operator creates an SEA Administrator. Every tenant must start with an SEA Administrator Account.
2. SEA Administrator creates a Realm Administrator and an Ingestion User.
3. Ingestion User prepares files for ingestion, provisions a landing zone, and ingests data.
4. Realm Administrator creates a realm.
5. SEA Administrator creates LEA Administrator in the Local Level Education Agency.
6. LEA Administrator verifies that default custom roles exist, and creates them as necessary.
7. LEA Administrator approves data browser application, or a third party application created for the district, to access data.
8. LEA Administrator logs in as a user and verifies that the user can view data through the dashboard and data browser.

On-boarding SIF Districts

Districts that use SIF data also on-boarding using bulk ingestion. However, SIF data must be converted into an Ed-Fi compliant format before it can be ingested to the SLI data store. Districts that use SIF data should work with CPSI to format and ingest their data.