# Table of Contents

Getting Data In .................................................................................................................. 1
Connectors .......................................................................................................................... 2
  Connector architecture ........................................................................................................ 2
    Hosted connectors .......................................................................................................... 2
    Remote connectors ......................................................................................................... 2
Platform versions .................................................................................................................. 2
Built-in connectors .............................................................................................................. 3
Connector logs ..................................................................................................................... 3
Managing Connectors .......................................................................................................... 4
  Installing a connector ........................................................................................................ 4
    Installing a connector as a bootstrap plugin ................................................................. 4
    Installing a connector using the Fusion UI ................................................................. 4
    Installing a connector using the Blob Store API ......................................................... 6
Updating a connector .......................................................................................................... 6
Deleting a connector .......................................................................................................... 7
  Deleting a connector using the Fusion UI ..................................................................... 7
  Deleting a connector using the REST API .................................................................. 7
Connector configuration ...................................................................................................... 8
  Configuring Connectors Using The API .................................................................... 8
  Configuring Connectors Using The Fusion UI ............................................................ 9
Using certificate-based encryption ...................................................................................... 9
Connector logs .................................................................................................................... 9
Remote Connectors ............................................................................................................. 9
  The connector client ...................................................................................................... 10
    Basic connector client usage ..................................................................................... 10
Known Issues ..................................................................................................................... 10
Developing Custom Connectors ......................................................................................... 11
Java Connector Development ............................................................................................. 11
  Java SDK configuration ............................................................................................... 11
  Plugin client .................................................................................................................. 13
  Java SDK security .......................................................................................................... 14
Connectors Security ............................................................................................................. 15
  Install OpenSSL binaries .............................................................................................. 15
  Set up the certificates ................................................................................................. 15
  Specify connectors-rpc system properties to provide the certs .................................. 17
The Random Content Connector ....................................................................................... 18
  Random Content Connector ....................................................................................... 18
  Quick start .................................................................................................................... 18
Collections .......................................................................................................................... 19
Auxiliary Collections .......................................................................................................... 19
System Collections ............................................................................................................. 21
Collection Configuration Properties ............................................................................... 21
**Getting Data In**

Data ingestion gets your data into Fusion Server, and data indexing stores it in a format that’s optimized for searching. These topics explain how to get your data into Fusion Server in a search-optimized format.

- *Collections* are a way of grouping data sets so that related data sets can be managed together. Every data set that you ingest belongs to a collection. Any app can contain one or more collections. See Collection Management.

- *Datasources* are configurations that determine how your data is handled during ingest by Fusion Server’s connectors, parsers, and index pipelines. When you run a fully-configured datasource, the result is an indexed data set that’s optimized for search, depending on the shape of your data and how you want to search it. See Datasource Configuration.

- In some cases, you might find that it’s best to use other ingestion methods, such as the Fusion Bulk Loader, Hive, Pig, or pushing data to a REST API endpoint.

- Blob storage is a way to upload binary data to Fusion Server. This can be your own data, such as images or executables, or it can be plugins for Fusion Server, such as connectors, JDBC drivers, and so on.
Connectors

Connectors are the built-in mechanism for pulling your data into Fusion. Lucidworks provides a wide variety of connectors, each specialized for a particular data type. When you add a datasource to a collection, you specify the connector to use for ingesting data. See the complete list of connectors, with links to configuration reference information for each one.

Connector architecture

As of Fusion Server 4.0, connector plugins can be hosted within Fusion, or can run remotely. The communication of messages between Fusion and a remote Connector or hosted Connector are identical; Fusion sees them as the same kind of Connector. This means you can implement a plugin locally, connect to a remote Fusion for initial testing, and when done, upload the same artifact into Fusion, so Fusion can host it for you.

The connectors architecture was designed to be scalable. Depending on the connector, jobs can now be scaled by adding new instances of the connector. The fetching process for these new types also supports distributed fetching, so that many instances can contribute to the same job.

Hosted connectors

In the hosted case, connectors are cluster aware. This means that when a new instance of Fusion starts up, the connectors on other Fusion nodes become aware of the new connectors, and vice versa. This makes scaling the crawling process very natural and simple.

Remote connectors

SDK connectors can be hosted within Fusion Server, or can run remotely. In the remote case, connectors become clients of Fusion. These clients run a very lightweight process and communicate to Fusion using a very efficient messaging format. This option makes it possible to put the connector wherever the data lives. This may be done for performance reasons, or for security/access reasons. See Remote Connectors for more details.

Platform versions

Initially, Fusion offered classic connectors, or V1, connectors. V1 connectors were developed with general-purpose crawler framework called Anda, created by Lucidworks. Anda helps simplify and streamline crawler development, reducing the task of developing a new crawler to gain access to your data.

As of version 4.1.0, Fusion began offering V2 connectors, which utilize a Java SDK framework.

The V2 platform version is included by default for all connectors it is available for. Currently, three connectors are offered with in the V2 platform version: Local Filesystem, OneDrive, and Sitecore. The Local Filesystem connector is also offered in V1 upon request. OneDrive and Sitecore connectors are only offered in V2, having been developed after the V2 platform version became available.

In addition to the features and benefits provided by V1 connectors, V2 connectors offer:

- Security Access-control Lists (ACL) which are separate from content
- Fusion connectors support SSL/TLS security
- Improved scalability, depending on the connector
Jobs can be scaled by simply adding instances of the connector
The fetching process supports distributed fetching, allowing many instances to contribute to the same job

Connectors can be hosted within Fusion, or can run remotely
Hosted connectors are cluster-aware, allowing connectors on separate notes to become of new connectors
Remote connectors become clients of Fusion and run a lightweight process and communicate to Fusion using an efficient messaging format
Remote connectors can be located wherever the data is located, which might be required for performance or security and access

Google's fast and efficient framework gRPC is used as the underlying client/server technology
Increased flexibility in the way services and their methods are defined
HTTP/2 based transport
Efficient serialization format for data handling (protocol buffers)
Allows bi-directional/multiplexed stream

Built-in connectors
Fusion comes with a standard set of built-in connectors:

- Local Filesystem connector
- File Upload connector
- JDBC connector
- Web connector

Additional connectors are available for download at http://lucidworks.com/connectors/. You can look in fusion/5.0.x/apps/connectors/connectors-classic/plugins/ and fusion/5.0.x/apps/connectors/connectors-rpc/plugins/ to see which additional connectors are currently installed.

Connector logs
You can find connector logs in fusion/5.0.x/var/log/connectors.

SDK connectors support Diagnostic Mode, which enables Fusion to print more detailed information to the logs about each request, including the ID of every document inserted, updated, or deleted in the oplog. More information on Diagnostic Mode can be found in the Configuration section of the connectors which offer it:

- MongoDB
- Windows Share SMB 2/3
- Confluence
- Salesforce
- ServiceNow
- Jive
Managing Connectors

Installing a connector

Connectors are installed by uploading them to the blob store. You can install connectors:

- By installing connectors as “bootstrap plugins”, that is, by putting them in the `bootstrap-plugins` directory during initial installation or an upgrade
- By using the Fusion UI after installation or an upgrade
- By using the Blob Store API after installation or an upgrade.

| Note | During upgrades, the migrator handles some aspects of installing connectors. Depending on the target version and the presence or absence of an Internet connection, there might be manual steps. Installing connectors during upgrades is explained where needed in the upgrade procedures. |

Installing a connector as a bootstrap plugin

Fusion can install connectors as “bootstrap plugins.” All this means is that you put the connector zip files in a specific directory named `bootstrap-plugins`, and Fusion installs the connectors the first time it starts during initial installation or an upgrade.

How to install a connector as a bootstrap plugin


   Don’t expand the archive; Fusion consumes it as-is. Also, don’t start Fusion until instructed to do so by the installation or upgrade instructions.

2. Under the version-numbered Fusion directory, place the connector in the directory `apps/connectors/bootstrap-plugins/` (on Unix) or `\apps\connectors\bootstrap-plugins\` (on Windows).

3. Start Fusion when instructed to do so in the installation or upgrade procedure.

Installing a connector using the Fusion UI


   Do not expand the archive; Fusion consumes it as-is.

2. In the Fusion workspace, navigate to System > Blobs.

3. Click Add.

4. Select Connector Plugin.
The "New Connector Plugin Upload" panel appears.

5. Click **Choose File** and select the downloaded zip file from your file system.

6. Click **Upload**.

The new connector's blob manifest appears.
From this screen you can also delete or replace the connector.

**Installing a connector using the Blob Store API**


   Do not expand the archive; Fusion consumes it as-is.

2. Upload the connector zip file to Fusion's blob store.

   Specify an arbitrary blob ID, and a `resourceType` value of `plugin:connector`, as in this example:

   ```bash
   curl -H 'content-type:application/zip' -X PUT 
   'localhost:8764/api/blobs/myplugin?resourceType=plugin:connector' --data-binary @myplugin.zip
   ```

   Fusion automatically publishes the event to the cluster, and the listeners perform the connector installation process on each node.

   **Tip**

   If the blob ID is identical to an existing one, the old connector will be uninstalled and the new connector will installed in its place. To get the list of existing blob IDs, run:

   ```bash
   curl -u user:pass localhost:8764/api/blobs
   ```

3. Look in `fusion/5.0.x/apps/connectors/plugins/` to verify that the new connector is installed.

**Updating a connector**

On Unix, you can update a connector by simply uploading the new one. Fusion overwrites the old one, and no restart is needed.

On Windows, a different procedure is needed:

**How to update a Fusion connector on Windows**

1. Delete the old connector, as explained below.

2. Restart Fusion.
3. Upload the new connector.

**Deleting a connector**

You can delete a connector using the Fusion UI or the Blob Store API.

**Deleting a connector using the Fusion UI**

1. In the Fusion UI, navigate to **System > Blobs**.
2. Under **Connector Plugin**, select the connector to delete.
3. Click **Delete Blob**.

   ![Deleting a connector using the Fusion UI](image)

   Fusion prompts you to confirm that you want to delete the blob.

4. Click **Yes, Delete**.

   The connector disappears from the blob list.

**Deleting a connector using the REST API**

1. Get the list of blobs of the connector plugin type:

   ```bash
   curl -u user:pass http://localhost:8764/api/blobs?resouType=plugin:connector
   ```

2. Locate the connector you want to delete, and copy its ID.

   For example, the Jive connector ID is **lucid.jive**:
3. Delete the connector as follows:

```bash
curl -u user:pass -X DELETE http://localhost:8764/api/blobs/<id>
```

For example

```bash
curl -u user:pass -X DELETE http://localhost:8764/api/blobs/lucid.jive
```

A null response indicates success. You can verify that the connector is deleted like this:

```bash
curl -u user:pass http://localhost:8764/api/blobs | grep lucid.jive
```

**Connector configuration**

When you add a datasource to a collection, you select a connector and configure it. There are two ways to do this:

- Using the API
- Using the UI

**Configuring Connectors Using The API**

You can create or update a datasource with the Connector Datasources API, specifying the connector, its properties, and their values.

Example: *Create and configure a datasource to index Solr-formatted XML files*

```bash
curl -u user:pass -X POST -H 'Content-type: application/json' -d '{"id":"SolrXML",
"connector":"lucid.solrxml", "type":"solrxml", "properties":{"path":"/Applications/solr-
4.10.2/example/exampledocs", "generate_unique_key":false, "collection":"MyCollection"}}' http://fusion-host:{api-port}/api/connectors/datasources
```

See the Connectors and Datasources Reference for details about configuration options.

| Tip | Be sure the include the collection property; otherwise the datasource will not be available in the Fusion UI. |
Example: Change the max_docs value for the above datasource

curl -u user:pass -X PUT -H 'Content-type: application/json' -d '{"id":"SolrXML", "connector":"lucid.solrxml", "type":"solrxml", "properties":{"path":"/Applications/solr-4.10.2/example/exampledocs", "max_docs":10}}' http://fusion-host:{api-port}/api/connectors/datasources/SolrXML

Configuring Connectors Using The Fusion UI

• To create and configure a new datasource and its connector:
  1. In the Fusion workspace, navigate to Indexing > Index Workbench.
  2. Click New.
  3. Select Or, if you know the location of your data.
  4. In the dropdown list, select the datasource type, which corresponds to a Fusion connector.
  5. Click Add New Datasource.
  6. Edit the configuration fields in the datasource panel that appears.
     See the Connectors and Datasources Reference for details about configuration options.
  7. Click Apply to test your configuration.
     Make changes as needed until the preview panel displays your data as it should be indexed.
  8. Click Save.

• To change the connector configuration for an existing datasource:
  1. In the Fusion workspace, navigate to Indexing > Index Workbench.
  2. Click Load and select the datasource you want to change.
  3. Edit the configuration fields as needed.
  4. Click Apply to test your configuration.
     Make changes as needed until the preview panel displays your data as it should be indexed.
  5. Click Save.

Using certificate-based encryption

Fusion connectors can be configured for use with certificate-based encryption, often used for signing certificates used in HTTPS connections. Certificates issued by the certificate authority must be placed in /data/fusion/5.0.x/apps/jetty/connectors-classic/etc/.

Connector logs

You can find connector logs in fusion/5.0.x/var/log/connectors.

Remote Connectors

The Fusion connector architecture is designed to be scalable. Depending on whether the connector is a V1 or a V2 (SDK) connector, jobs can be scaled by adding new instances of just the connector. The fetching process for these connectors
also supports distributed fetching, so that many instances can contribute to the same job.

| Important | At this time, Fusion 5.0.x does not support remote connectors. |

SDK connectors can be hosted within Fusion Server, or can run remotely. In the hosted case, these connectors are cluster aware. This means that when a new instance of Fusion starts up, the connectors on other Fusion nodes become aware of the new connector, and vice versa. This makes scaling connector jobs simple.

In the remote case, a connector becomes a client of Fusion. This remote client runs a lightweight process and communicates to Fusion using an efficient messaging format. This option makes it possible to put the connector wherever the data lives. This can be done for performance reasons, or for security or access reasons.

The default SDK connector service is `connectors-rpc`. By default, `connectors-rpc` runs on port 8771. This service handles connector registration, configuration management, job management, and cluster coordination. Like other Fusion services, it also provides access to non-connector clients.

### The connector client

Fusion comes with a connector client that remote connectors can use to communicate with Fusion. It is located at `FUSION_HOME/apps/connectors/connectors-rpc/client/connector-plugin-client-{fusionVersion}.x-uberjar.jar`.

To run the connector client, you must have a `.zip` file containing exactly one connector plugin. Visit our [Connector Downloads](#) page to obtain a copy of the available V2 connectors.

#### Basic connector client usage

To start a connector client, on the remote node (for example, the datasource), do the following:

1. Copy the connector uberjar from Fusion Server onto the remote node. The connector uberjar is at the following location:

   FUSION_HOME/apps/connectors/connectors-rpc/client/connector-plugin-client-{fusionVersion}-uberjar.jar

2. On the remote node, run:

   ```
   java -jar path/to/uberjar/connector-plugin-client-{fusionVersion}-uberjar.jar path/to/connector/file.zip
   ```

### Known Issues

- Registering a plugin instance during crawl could result in errors. Only connect plugins when no jobs are running.

- In order to connect a plugin from a remote instance, you are required to manually set the `default.address` value in Fusion. This host value is used with the property `com.lucidworks.fusion.plugin.hosts`. For example, where 10.10.10.10 is the host value in the `FUSION_HOME/conf/fusion.properties` file:

   ```
   java -Dcom.lucidworks.fusion.plugin.hosts=10.10.10:8771 -jar path/to/uberjar/connector-plugin-client-{version}-uberjar.jar path/to/connector/file.zip
   ```
Developing Custom Connectors

Lucidworks provides a Connectors SDK in a public repository on GitHub with all the resources you need to develop custom connectors. Clone the repository to get started:

```
git clone https://github.com/lucidworks/connectors-sdk-resources
```

It includes the Random Content connector, which generates random documents and serves as an example project that you can use as a model for your own custom connector projects.

Currently, only Java is supported. See the Javadocs.

These topics provide details about developing custom connectors for Fusion:

- Java Connector Development explains the connector framework and how to develop a connector in Java.
- Connectors Security describes how to set up OpenSSL and certificates.
- The Random Content Connector shows you an example connector that generates random documents.

Java Connector Development

Java SDK configuration

To build a valid connector configuration, you must:

- Define an interface.
- Extend `ConnectorConfig`.
- Apply a few annotations.
- Define connector "getter" methods and annotations.

All methods that are annotated with `@Property` and start with "get" are considered to be configuration properties. For example, `@Property() String getName();` results in a String property called `name`. This property would then be present in the generated schema.

Here is an example of the most basic configuration, along with required annotations and a sample "getter" method:
The metadata defined by \texttt{@RootSchema} is used by Fusion when showing the list of available connectors. The \texttt{ConnectorConfig} base interface represents common, top-level settings required by all connectors. The \texttt{type} parameter of the \texttt{ConnectorConfig} class indicates the interface to use for custom properties.

Once a connector configuration has been defined, it can be associated with the \texttt{ConnectorPlugin} class. From that point, the framework takes care of providing the configuration instances to your connector. It also generates the schema, and sends it along to Fusion when it connects to Fusion.

\textbf{About com.lucidworks.schema}

The \texttt{com.lucidworks.schema} project (included in the connectors SDK) aims to simplify the process of creating JSON Schemas. It also provides utilities for building type-safe POJOs from schema definitions.

The basic idea here is that you create an interface that extends \texttt{Model}, add property getters, and then add a few annotations. Once your interface has been built, you can generate an \texttt{ObjectType} instance, which is the object that contains all of the JSON schema metadata. By then combining that schema object with a \texttt{Map<String, Object>}, you can create instances of your interface to use as configuration objects.

The configuration objects are based on \texttt{java.lang.reflect.Proxy}, which is what the \texttt{ModelGenerator} returns. Most method calls to these proxy instances result in method calls to the underlying \texttt{Map<String, Object>}. For example, if the interface defines a \texttt{getId} method, then a runtime call results in a call to the \texttt{Map<String, Object>}: \texttt{data.get("id")}.

A few special cases exist:

- \texttt{toString} is proxied directly to the \texttt{Map<String, Object>}

```java
@RootSchema(
    name = "com.my.connector",
    title = "My Connector",
    description = "My Connector description",
    category = "My Category"
)
public interface MyConfig extends ConnectorConfig<MyConfig.Properties> {

    @Property(
        title = "Properties",
        required = true
    )
    public Properties getProperties();

    /**
     * Connector specific settings
     */
    interface Properties extends FetcherProperties {

        @Property(
            title = "My custom property",
            description = "My custom property description"
        )
        public Integer getMyCustomProperty();
    }
}
```
• `getClass` returns the class of the interface provided to `ModelGenerator#generate()`

• `_data` returns the underlying `Map<String, Object>` object

• If the method starts with `set`, the arguments are captured and sent to the underlying map via `put()`

Here is a simple example:

```java
interface MyConfig extends Model {
    @Property @StringSchema(minLength=1)
    String getId();
}
```

Create the schema:

```java
class Runner {
    public static void main(String[] args){
        ObjectType schema = SchemaGenerator.generate(MyConfig.class);
    }
}
```

Generate an instance of the `MyConfig` class:

```java
class Runner {
    public static void main(String[] args){
        Map<String, Object> data = new java.util.HashMap<>();
        data.put("id", 100);
        MyConfig config = ModelGenerator.generate(MyConfig.class, data);
        System.out.println("id -> " + config.getId());
    }
}
```

Schema metadata can be applied to properties using additional annotations. For example, applying limits to the min/max length of a string, or describing the types of items in an array.

Nested schema metadata can also be applied to a single field by using "stacked" schema based annotations:

```java
interface MySetConfig extends Model {
    @SchemaAnnotations.Property(title = "My Set")
    @SchemaAnnotations.ArraySchema(defaultValue = "["a"]")
    @SchemaAnnotations.StringSchema(defaultValue = "some-set-value", minLength = 1, maxLength = 1)
    Set<String> getMySet();
}
```

**Plugin client**

The Fusion connector plugin client provides a wrapper for the Fusion Java plugin-sdk so that plugins do not need to directly talk with gRPC code. Instead, they can use high-level interfaces and base classes, like Connector and Fetcher.
The plugin client also provides a standalone "runner" that can host a plugin that was built from the Fusion Java Connector SDK. It does this by loading the plugin zip file, then calling on the wrapper to provide the gRPC interactions.

### Java SDK / gRPC Wrapper

One of the primary goals of the plugin-client is to isolate plugin code from the underlying framework details. Specifically, the underlying message formats (protocol buffers) and native gRPC code. This makes it possible to make some changes to the base support layer, without having to make changes to the Java plugin implementation.

### Standalone Connector Plugin Application

The second goal of the plugin-client is to allow Java SDK plugins to run remotely. The instructions for deploying a connector using this method are provided below.

#### Locating the UberJar

The uberjar is located in this location in the Fusion file system:

```
$FUSION_HOME/apps/connectors/connectors-rpc/client/connector-plugin-client-<version>-uberjar.jar
```

where `$FUSION_HOME` is your Fusion installation directory and `<version>` is your Fusion version number.

#### Starting the Host

To start the host app, you need a Fusion SDK-based connector, built into the standard packaging format as a `.zip` file. This `zip` must contain only one connector plugin.

Here is an example of how to start up using the web connector:

```
java -jar $FUSION_HOME/apps/connectors/connectors-rpc/client/connector-plugin-client-<version>-uberjar.jar
fusion-connectors/build/plugins/connector-web-4.0.0-SNAPSHOT.zip
```

To run the client with remote debugging enabled:

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5010 -jar
$FUSION_HOME/apps/connectors/connectors-rpc/client/connector-plugin-client-<version>-uberjar.jar
fusion-connectors/build/plugins/connector-web-4.0.0-SNAPSHOT.zip
```

#### Java SDK security

See the main custom connector security overview for instructions on how to configure SSL/TLS support for a Fusion custom connector.

### Java Plugin Client

The information here is specific to running a Java SDK plugin outside of Fusion.

The plugin-client supports several variations of SSL/TLS auth. The examples below show the relevant Java properties.

#### Example with Mutual TLS auth and private key passwords

...
Connectors Security

This topic helps you set up transport layer security (TLS) between your Connectors-rpc server (using gRPC) and Connector JVM.

This is recommended in a production environment.

Install OpenSSL binaries

The gRPC framework makes TLS more efficient by using native openssl binaries while doing SSL. To take advantage of this efficiency, install OpenSSL for your operating system.

**Install OpenSSL on Ubuntu or Debian**

```
sudo apt-get install openssl
```

**Install OpenSSL on CentOS, Redhat Linux, or Amazon EC2**

```
sudo yum install openssl
```

**Install OpenSSL on Windows**

1. Install OpenSSL, for example from [https://slproweb.com/](https://slproweb.com/).
2. Add the installed binaries to your path.

Set up the certificates

In a production environment, use an SSL certificate from a trusted certificate authority. For testing purposes, you can use a self-signed certificate. Generate a self-signed certificate with the following steps.

Once you have the SSL certificates in the server.key and server.pem files, continue to the Specify rpc-system properties step.

Linux setup

Create a new folder, open a terminal, cd to the new folder, then run this bash script:
# Change these CNs to match your hosts in your environment if needed.
SERVER_CN=myhost
CLIENT_CN=myhost # Used when doing mutual TLS

echo Generate CA key:
openssl genrsa -passout pass:1111 -des3 -out ca.key 4096
echo Generate CA certificate:
# Generates ca.crt, the trustCertCollectionFile
openssl req -passin pass:1111 -new -x509 -days 365 -key ca.key -out ca.crt -subj "/CN=${SERVER_CN}"
echo Generate server key:
openssl genrsa -passout pass:1111 -des3 -out server.key 4096
echo Generate server signing request:
openssl req -passin pass:1111 -new -key server.key -out server.csr -subj "/CN=${SERVER_CN}"
echo Self-signed server certificate:
# Generates server.crt, the certChainFile for the server
openssl x509 -req -passin pass:1111 -days 365 -in server.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out server.crt
echo Remove passphrase from server key:
openssl rsa -passin pass:1111 -in server.key -out server.key
echo Generate client key
openssl genrsa -passout pass:1111 -des3 -out client.key 4096
echo Generate client signing request:
openssl req -passin pass:1111 -new -key client.key -out client.csr -subj "/CN=${CLIENT_CN}"
echo Self-signed client certificate:
# Generates client.crt, the clientCertChainFile for the client (needed for mutual TLS only)
openssl x509 -passin pass:1111 -req -days 365 -in client.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out client.crt
echo Remove passphrase from client key:
openssl rsa -passin pass:1111 -in client.key -out client.key
echo Converting the private keys to X.509:
# Generates client.pem, the clientPrivateKeyFile for the Client (needed for mutual TLS only)
openssl pkcs8 -topk8 -nocrypt -in client.key -out client.pem
# Generates server.pem, the privateKeyFile for the Server
openssl pkcs8 -topk8 -nocrypt -in server.key -out server.pem

Windows setup

The default installation directory is C:\OpenSSL-Win64

Create a new directory, open a cmd shell, cd to that directory, and run this batch:
@echo off
set OPENSSL_CONF=c:\OpenSSL-Win64\bin\openssl.cfg

# Change these CNs to match your hosts in your environment if needed.
set SERVER_CN=myhost
set CLIENT_CN=myhost # Used when doing mutual TLS

echo Generate CA key:
openssl genrsa -passout pass:1111 -des3 -out ca.key 4096

echo Generate CA certificate:
# Generates ca.crt, the trustCertCollectionFile
openssl req -passin pass:1111 -new -x509 -days 365 -key ca.key -out ca.crt -subj "/CN=%SERVER_CN%"

echo Generate server key:
openssl genrsa -passout pass:1111 -des3 -out server.key 4096

echo Generate server signing request:
openssl req -passin pass:1111 -new -key server.key -out server.csr -subj "/CN=${SERVER_CN}"

echo Self-signed server certificate:
# Generates server.crt, the certChainFile for the server
openssl x509 -req -passin pass:1111 -days 365 -in server.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out server.crt

echo Remove passphrase from server key:
openssl rsa -passin pass:1111 -in server.key -out server.key

echo Generate client key
openssl genrsa -passout pass:1111 -des3 -out client.key 4096

echo Generate client signing request:
openssl req -passin pass:1111 -new -key client.key -out client.csr -subj "/CN=${CLIENT_CN}"

echo Self-signed client certificate:
# Generates client.crt, the clientCertChainFile for the client (need for mutual TLS only)
openssl x509 -req -days 365 -in client.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out client.crt

echo Remove passphrase from client key:
openssl rsa -passin pass:1111 -in client.key -out client.key

echo Converting the private keys to X.509:
# Generates client.pem, the clientPrivateKeyFile for the Client (needed for mutual TLS only)
openssl pkcs8 -topk8 -nocrypt -in client.key -out client.pem
# Generates server.pem, the privateKeyFile for the Server
openssl pkcs8 -topk8 -nocrypt -in server.key -out server.pem</td>

Specify connectors-rpc system properties to provide the certs

Now that we have the certs, we set them in the system properties.

Example with mutual TLS auth and private key passwords

-Dcom.lucidworks.apollo.app.hostname=myhost
-Dcom.lucidworks.fusion.tls.server.certChain=./sslcerts/server.crt
-Dcom.lucidworks.fusion.tls.server.privateKey=./sslcerts/server.pem
-Dcom.lucidworks.fusion.tls.server.privateKeyPassword=password123
-Dcom.lucidworks.fusion.tls.clientCertChain=./sslcerts/client.crt
-Dcom.lucidworks.fusion.tls.requireMutualAuth=false

Example without TLS auth and no private key passwords
The Random Content Connector

The Connectors SDK comes with a sample connector to help you.

Random Content Connector

The Random Content connector generates a configurable number of documents with random titles and body fields.

Quick start

<table>
<thead>
<tr>
<th>Note</th>
<th>This quick start assumes that Fusion is installed on the /opt path.</th>
</tr>
</thead>
</table>

1. Clone the repo:

```bash
git clone https://github.com/lucidworks/connectors-sdk-resources.git
cd connectors-sdk-resources/java-sdk/connectors/
./gradlew assemblePlugins
```

1. This produces one zip file, named `random-connector.zip`, located in the `build/plugins` directory. This artifact is now ready to be uploaded directly to Fusion as a connector plugin.

2. See the following [instructions](../README.md) on how to build, deploy, and run the plugi
Collections

Your data is organized into collections. When you create an app, Fusion automatically creates a collection with the same name. You can create additional collections in any app.

A primary collection contains the data that your users will search. Every primary collection is associated with a set of auxiliary collections that contain related data, such as signals, aggregations, and more.

Under the hood, a Fusion collection is a distributed index in Solr, defined by a named configuration stored in ZooKeeper, with these properties:

- Number of shards
  Documents are distributed across this number of partitions.

- Document routing strategy
  How documents are assigned to shards.

- Replication factor
  How many copies of each document in the collection.

- Replica placement strategy
  Where to place replicas in the cluster.

If your data is already stored in a Solr instance or cluster, you can manage this collection in Fusion by creating a Fusion collection that imports the existing Solr collection. See Installation with an existing Solr instance or cluster.

| Note | Collection names are case-insensitive, but Fusion preserves case when displaying collection names. |

Auxiliary Collections

Every primary collection is associated with a set of auxiliary collections that contain related data, such as signals, aggregations, and more.

Some auxiliary collections are created for every primary collection. Others are created only for the app's default collection, one per app.

Auxiliary collections are described below:

<p>| &lt;App&gt;_job_reports | Output from Fusion AI experiments, Ranking Metrics jobs, and Head/Tail Analysis jobs. | 1 per app |</p>
<table>
<thead>
<tr>
<th>Collection Name</th>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
</table>
| `<App>_query_rewrite` | A collection of documents to use for rewriting queries, optimized for high-volume traffic. These documents originate from the `_query_rewrite_staging` collection. Certain Fusion AI query pipeline stages read from this collection:  
  - Text Tagger  
  - Apply Rules  
  - Modify Response with Rules | 1 per app |
| `<App>_query_rewrite_staging` | A collection of documents created by the Rules Editor or by certain Fusion AI jobs, not optimized for production traffic.  
  Documents move from this collection to the `_query_rewrite` collection as follows:  
  - Job output documents with high confidence contain a `review=auto` field and are moved to the `_query_rewrite` collection automatically.  
  - Job output documents with low confidence contain a `review=pending` field. When these are approved by a Fusion user, Fusion copies them to the `_query_rewrite` collection. | 1 per app |
| `<Collection>_signals` | A search query logs and signals collection. | 1 per collection |
| `<Collection>_signals_aggr` | A collection for aggregated signals. | 1 per collection |
| `<App>_user_prefs` | A collection of data to support App Studio's social features, such as user-generated tags, bookmarks, comments, ratings, and so on. | 1 per app |
Fusion maintains a set of Solr collections that store Fusion’s own log files and other internal information. These are called System Collections, described below.

Fusion uses ZooKeeper to register information about all collections, and the Fusion components and services related to a collection. The Fusion components associated with a collection include:

- Datasources
- Pipelines
- Profiles
- Signals and aggregations
- Analytics dashboards

## System Collections

Fusion automatically creates some collections that are used for internal purposes and shared across all apps:

- **system_autocomplete** store the content that the Fusion UI displays when you use the search bar.
- **system_blobs** stores blobs in Solr. This is used to store model files for the NLP components and other binary files used by Fusion components.
- **system_history** keeps a record of configuration changes, start and stop times for services and experiments, and more.
- **system_jobs_history** keeps a record of Fusion jobs, including start/stop times and status.
- **system_logs** stores parsed Java, logs from the REST API, connectors-classic component, and other parts of Fusion, like proxy, connectors-rpc, and appkit app insights. It also includes http logs and optional gc logs (off by default in Fusion 4.1). Prior to Fusion version 4.1, Java logs were stored in the `logs` collection and HTTP requests were stored in the `audit_logs` collection.
- **system_messages** is used by Fusion’s messaging services.
- **system_monitor** stores metrics about Fusion hosts and services. See System Metrics and the DevOps Center.

## Collection Configuration Properties

Collections have three properties that you can configure only when you are creating a collection using the Collections API.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>signals*</td>
<td>The <code>signals</code> property determines whether to create auxiliary collections with suffixes <code>_signals</code> and <code>_signals_aggr</code>.</td>
<td>When you create a collection in the Fusion UI, <code>signals</code> defaults to <code>true</code>. When you create a collection using the Fusion API, this property defaults to <code>false</code>.</td>
</tr>
<tr>
<td>searchLogs</td>
<td>The <code>searchLogs</code> property determines whether to create an auxiliary search query logs collection with suffix <code>_logs</code>.</td>
<td>When you create a collection in the Fusion UI, this property defaults to <code>true</code>. When you create a collection using the Fusion API, this property defaults to <code>false</code>.</td>
</tr>
<tr>
<td>dynamicSchema</td>
<td>When <code>dynamicSchema</code> is <code>true</code>, Fusion and Solr use schemaless mode to administer search and indexing over that collection.**</td>
<td>Property <code>dynamicSchema</code> always defaults to <code>false</code>.</td>
</tr>
</tbody>
</table>

*Signals are events with timestamps that can be used to improve search results. For more information about signals in Fusion, see Signals in the Fusion AI documentation.

**In schemaless mode, if a document contains a field not currently in the Solr schema, Solr processes the field value to determine what the field type should be defined as, and then adds a new field to the schema with the field name and field type. This behavior can be convenient during preliminary application development, but it is rarely appropriate in a production environment.

Using profiles to associate collections with pipelines

Index pipelines and query pipelines are not connected to a specific collection by default. Index profiles and query profiles are configurations that create consistent endpoints for indexing and querying, each with a specific pipeline and collection.

- Index Profiles work with index pipelines for getting content into the system.
- Query Profiles work with query pipelines for user queries.

Collection Management in the Fusion UI

Collections can be created or removed using the Fusion UI or the REST API. For information about using the REST API to manage collections, see Collections API in the REST API Reference.

Creating a Collection

When you create an app, by default Fusion Server creates a collection and associated objects.

To create a new collection in the Fusion UI:
1. From within an app, click **Collections > Collections Manager**.
2. At the upper right of the panel, click **New**.
3. Enter a **Collection name**. This name cannot be changed later.
4. To create the collection in the default Solr cluster and with other default settings, click **Save Collection**.

### Creating a Collection with Advanced Options

To access advanced options for creating a collection in the Fusion UI:

1. From within an app, click **Collections > Collections Manager**.
2. At the upper right of the panel, click **New**.
3. Enter a **Collection name**. This name cannot be changed later.
4. Click **Advanced**.
5. Configure advanced options. The options are described below.
6. Click **Save Collection**.

### Solr Cluster

By default, a new collection is associated with the Solr instance that is associated with the **default** Solr cluster.

If Fusion has multiple Solr clusters, choose from the list which cluster you want to associate your collection with. The cluster must exist first.

See Search Cluster API in the REST API Reference for information about connecting Fusion with an existing Solr instance.

### Solr Cluster Layout

The next section lets you define a **Replication Factor** and **Number of Shards**. Define these options only if you are creating a new collection in the Solr cluster. If you are linking Fusion to an existing Solr collection, you can skip these settings.

### Solr Collection Import

Import a Solr collection to associate the new Fusion collection with an existing Solr collection. Enter a **Solr Collection Name** to associate the collection with an existing Solr collection. Then, enter a **Solr Config Set** to tell ZooKeeper to use the configurations from an existing collection in Solr when creating this collection.

### Time Series Partitioning

You can map a Fusion collection to multiple Solr collections, known here as partitions, where each partition contains data from a specific time range.

To configure time-based partitioning, under **Time Series Partitioning** click **Enable**.

See Time-Based Partitioning for configuration options.

### Configuring Collections

The Collections menu lets you configure your existing collection, including datasources, fields, jobs, stopwords, and
In the Fusion UI, from any app, find the Collections menu on the left side of the screen.

Some tasks related to managing a collection are available in other menus:

- Configure a profile in Indexing > Indexing Profiles or Querying > Query Profiles.
- View reports about your collection's activity in Analytics > Dashboards.

**Collections Manager**

The Collections Manager page displays details about the collection, such as how many datasources are configured, how many documents are in the index, and how much disk space the index consumes.

This page also lets you create a new collection, disable search logs or signals, enable recommendations, issue a commit command to Solr, or clear a collection.

**Disable search logs**

When you first create a collection, the search logs are created by default. The search logs populate the panels in Analytics > Dashboards.

To disable search logs:

1. Hover over your collection name until the gear icon appears at the end of the line.
2. Click the gear icon.
3. Click Disable Search Logs.
4. On the confirmation screen, click Disable Search Logs.

Note that if you disable search logs, you cannot see any data for this collection in Analytics > Dashboards. See Dashboards in the System Administration Guide.

**Disable signals**

When you first create a collection, the signals and aggregated signals collections are created by default. To make use of the signals in your collection, use Fusion AI.

To disable signals:

1. Hover over your collection name until the gear icon appears at the end of the line.
2. Click the gear icon.
3. Click Disable Signals.
4. On the confirmation screen, click Disable Signals.

**Hard commit a collection**

To issue a hard commit command to Solr:

1. Hover over your collection name until the gear icon appears at the end of the line.
2. Click the gear icon.
3. Click **Hard Commit Collection**.
4. On the confirmation screen, click **Hard Commit Collection**.

Read internal details about how Solr processes commits on [our blog](#).

**Datasources**

To access the Datasources page, click **Indexing > Datasources**. By default, there are no datasources configured right after installation.

To add a new datasource, click **New** at the upper right of the panel. See the Connectors and Datasources Reference for details on how to configure a datasource. Options vary depending on the repository you would like to index.

After you configure a datasource, it appears in a list on this screen. Click the name of a datasource to edit its properties. Click **Start** to start the datasource. Click **Stop** to stop the datasource before it completes. To the right, view information on the last completed job, including the date and time started and stopped, and the number of documents found as new, skipped, or failed.

| Note | When you stop a datasource, Fusion attempts to safely close connector threads, finishing processing documents through the pipeline and indexing documents to Solr. Some connectors take longer to complete these processes than others, so might stay in a "stopping" state for several minutes. |

To stop a datasource immediately, choose **Abort** instead of **Stop**.

There is also a REST API for datasources; see the section Connector Datasources API for details.

**Stopwords**

The Stopwords page lets you edit a stopwords list for your collection.

To add or delete stop words:

1. Click the name of the text file you wish to edit.
2. Add a new word on a new line.
3. When you are done with your changes, click **Save**.

To import a stop words list:

1. Click **System > Import Fusion Objects**.
2. Choose the file to upload.
3. Click **Import >>**.

**Synonyms**

Fusion has the same synonym functionality that Solr supports. This includes a list of words that are synonyms (where the synonym list expands on the terms entered by the user), as well as a full mapping of words, where a word is substituted for what the user has entered (that is, the term the user has entered is replaced by a term in the synonym list). See the Apache Solr Reference Guide section on [the Synonym Filter](#) for more details.
You can edit the synonyms list for your collection.

To access the Synonyms page in the Fusion UI, in any app, click **Collections > Synonyms**.

Filter the list of synonym definitions by typing in the **Filter**... box.

To import a synonyms list:

1. From the Synonyms page, click **Import and Save**. A dialog box opens.
2. Choose the file to import.

To edit a synonyms list:

- Enter new synonym definitions one per line.
  - To enter a string of terms that expand on the terms the user entered, enter the terms separated by commas, like *Television, TV*.
  - To enter a term that should be mapped to another term, enter the terms separated by an equal sign then a right angle bracket, ⇒, like *i-pod⇒ipod*.
- Remove a line by clicking the **x** at the end of the line.
- Once you are finished with edits, click **Save**.

To export the synonyms list, click **Export**. This downloads the list to your computer using your browser download capability.

**Profiles**

Profiles allow you to create an alias for an index or query pipeline. This allows you to send documents or queries to a consistent endpoint and change the underlying pipeline or collection as needed.

Read about profiles in Index Profiles and Query Profiles.

To access the Solr Config page, from any app, click **System > Solr Config**.

**Time-Based Partitioning**

A Fusion collection can be configured to map to multiple Solr collections, known as partitions in this context, where each partition contains data from a specific time range. An example is time-based partitioning for logs:
Once a collection is configured for time-base partitioning, Fusion automatically ages out old partitions and creates new ones, using the configured partition sizes, expiration intervals, and so on. No manual maintenance is needed.

This feature is not enabled by default. Enable it for each collection using the Collection Features API.

| Note | Fusion cannot retroactively partition data that has already been indexed. It can only perform time-based partitioning on incoming data. |

**Enabling time-based partitioning**

- In the UI, you can only enable time-based partitioning for *new* collections.
- In the API, you can only enable time-based partitioning for *existing* collections.

**Enablement using the Fusion UI**

1. Open the Collections Manager:
2. Click New.

| Note | In the UI, you can only enable time-based partitioning for new collections. To enable it for an existing collection, use the API. |

3. Click Advanced.
4. Scroll down to "Time Series Partitioning".

5. Click **Enable**.

When you enable this option, Fusion displays the time series partitioning configuration options.

6. Save the collection.

**Enablement using the API**

Use the Collection Features API to enable time-based partitioning for an existing collection.

Enable time-based partitioning using the default configuration:
curl -X PUT -H 'Content-type: application/json' -d '{"enabled": true}'
http://localhost:8764/api/collections/<collection>/features/partitionByTime

No response is returned.

Submit an empty request to the same endpoint to verify that time-based partitioning is enabled:

curl -X GET http://localhost:8764/api/collections/<collection>/features/partitionByTime

Response:

```
{
  "name": "partitionByTime",
  "collectionId": "<collection>",
  "params": {},
  "enabled": true
}
```

To change the configuration, see the options and examples below.

**Configuration options**

When time series indexing is enabled for a collection, you can configure these options using the UI or the Collections API. None are required.

<table>
<thead>
<tr>
<th>UI Label, API Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp Field Name</td>
<td>The name of the field from which to read timestamps. The default is &quot;timestamp&quot;.</td>
</tr>
<tr>
<td>timestampFieldName</td>
<td></td>
</tr>
<tr>
<td>Partition Time Period</td>
<td>The time range for each partition. The default is one day.</td>
</tr>
<tr>
<td>timePeriod</td>
<td></td>
</tr>
<tr>
<td>Max Active Partitions</td>
<td>The number of partitions to keep active.</td>
</tr>
<tr>
<td>maxActivePartitions</td>
<td></td>
</tr>
<tr>
<td>Delete Expired Partitions</td>
<td>&quot;True&quot; to automatically delete partitions that fall outside of the maxActivePartitions window, at intervals of scheduleIntervalMinutes. The default is &quot;false&quot;.</td>
</tr>
<tr>
<td>deleteExpired</td>
<td></td>
</tr>
<tr>
<td>Preemptive Create Enabled</td>
<td>&quot;True&quot; (the default) to create partitions in advance.</td>
</tr>
<tr>
<td>preemptiveCreateEnabled</td>
<td></td>
</tr>
<tr>
<td>UI Label, API Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Schedule Interval</td>
<td>The interval, in minutes, at which to perform background maintenance, including preemptively creating partitions (preemptiveCreateEnabled) and deleting expired partitions (deleteExpired). The default is five minutes.</td>
</tr>
<tr>
<td>numShards</td>
<td>The number of shards per partition. The default is the value configured for the main Fusion collection.</td>
</tr>
<tr>
<td>replicationFactor</td>
<td>The number of copies to keep, per partition. The default is the value configured for the main Fusion collection.</td>
</tr>
<tr>
<td>configName</td>
<td>The name of the Solr configuration set to be applied to new partitions; the default is the configuration used by the primary collection.</td>
</tr>
</tbody>
</table>

**Examples**

Create a new collection called "TimeSeries1":

```
curl -X PUT -H 'Content-type: application/json' -d '{
"solrParams": {
    "numShards": 1,
    "replicationFactor": 1
}
}' http://localhost:8764/api/collections/TimeSeries1
```

Enable and configure time-based partitioning for the "TimeSeries1" collection:

```
curl -X PUT -H 'Content-type: application/json' -d '{
    "enabled": true,
    "timestampFieldName": "ts",
    "timePeriod": "5MINUTES",
    "scheduleIntervalMinutes": 1,
    "preemptiveCreateEnabled": false,
    "maxActivePartitions": 4,
    "deleteExpired": true
}' http://localhost:8764/api/collections/TimeSeries1/features/partitionByTime
```

Verify that time-based partitioning is enabled:

```
curl -X GET http://localhost:8764/api/collections/TimeSeries1/features/partitionByTime
```

Import some sample data into this collection:
curl -X POST -H "Content-type:application/vnd.lucidworks-document" -d '[
{
   "id": "1",
   "fields": [
   {
      "name": "ts",
      "value": "2016-02-24T00:00:01Z"
   },
   {
      "name": "partition_s",
      "value": "eventsim_2016_02_24_00_00"
   }
   ]
},
{
   "id": "2",
   "fields": [
   {
      "name": "ts",
      "value": "2016-02-24T00:05:01Z"
   },
   {
      "name": "partition_s",
      "value": "eventsim_2016_02_24_00_05"
   }
   ]
},
{
   "id": "3",
   "fields": [
   {
      "name": "ts",
      "value": "2016-02-24T00:10:01Z"
   },
   {
      "name": "partition_s",
      "value": "eventsim_2016_02_24_00_10"
   }
   ]
},
{
   "id": "4",
   "fields": [
   {
      "name": "ts",
      "value": "2016-02-24T00:15:01Z"
   },
   {
      "name": "partition_s",
      "value": "eventsim_2016_02_24_00_15"
   }
   ]
},
{
   "id": "5",
   "fields": [
   {
      "name": "ts",
      "value": "2016-02-24T00:20:01Z"
   },
   {
      "name": "partition_s",
      "value": "eventsim_2016_02_24_00_20"
   }
   ]
}]
'}
View the Solr configuration for this collection:


The response includes a list of active Solr collections that correspond to this Fusion collection:

<str name="collection">TimeSeries1_2016_02_24_00_05,TimeSeries1_2016_02_24_00_10,TimeSeries1_2016_02_24_00_15,TimeSeries1_2016_02_24_00_20</str>
Datasources

A collection includes one or more datasources. A datasource is a configuration that manages the import and indexing of data into the collection.

The Index Workbench provides a development environment for creating, configuring, and testing a datasource configuration. Every datasource configuration includes the following:

- Connector configuration, specifying the source and format of the incoming data.
- Parser configuration, describing a series of conditional parsing stages to transform the incoming data into PipelineDocument objects.
- Index pipeline configuration, consisting of stages that transform PipelineDocument objects into Solr documents to be indexed.

Collections and datasources can also be managed through the REST API.

In some cases it may make sense to bypass the connectors and use other ingest methods for your data.

Indexing Data

When a connector ingests data, it passes the data through an index pipeline for transformation prior to indexing. The format of your indexed data depends on the index pipeline, which is part of the datasource configuration.

Index pipelines

An index pipeline consists of one or more configurable index pipeline stages, each performing a different type of transformation on the incoming data. Each connector has a default index pipeline, but you can modify these or create new ones.

The last stage in any index pipeline should be the Solr Indexer stage, which submits the documents to Solr for indexing.
**Index profiles**

In some cases, you might want to send data to be indexed to an index profile, instead of directly to an index pipeline.

An index profile is an indexing endpoint that hides the index pipeline and collection from a search app, and that can be used to specify configuration parameters for the index pipeline.

**Index Pipelines**

Index pipelines transform incoming data into PipelineDocument objects for indexing by Fusion-managed Solr service. An index pipeline consists of a series of configurable index pipeline stages, each performing a different transformation on the data before passing the result to the next stage in the pipeline. The final stage is the Solr Indexer stage, which transforms the PipelineDocument into a Solr document and submits it to Solr for indexing in a specific Collection.

Each configured datasource has an associated index pipeline and uses a connector to fetch data to parse and then input into the index pipeline.

Alternatively, documents can be submitted directly to an index pipeline or profile with the REST API; see Importing Data with the REST API.

A pipeline can be reused across multiple collections. Fusion provides a set of built-in pipelines. You can use the Index Workbench or the REST API to develop custom index pipelines to suit any datasource or application.
When a Fusion collection is created using the Fusion UI, a pair of index and query pipelines are created to that pipeline, where the pipeline name is the collection name with the suffix "-default". This pipeline consists of a Field Mapping index stage.

Although default pipelines are created when a Fusion collection is created, they are not deleted when the collection is deleted. This is because pipelines can be used across collections, so a named pipeline, although originally associated with a collection, can be used by several collections.

Index Profiles

Index profiles let your applications send documents for indexing to a consistent endpoint (the index profile endpoint) and change the underlying index pipeline as needed.

An index profile is also a simple way to use one pipeline for multiple collections, without any one collection "owning" the pipeline. Associating a profile with an index pipeline, or an index pipeline and a collection, is simply a mapping.

Finally, you can also specify configuration parameters for an index pipeline in the index profile.

You can use the API or UI to manage index profiles.
Index Profiles in the API

• Index Profiles API (/index-profiles)
  Create, read, update, and delete index profiles.

• Index API (/index)
  Index data through an index profile by specifying the profile ID.

Index Profiles in the UI

Index profiles are configured at Indexing > Index Profiles.

How to create a new index profile

1. In the Fusion workspace, navigate to Indexing > Index Profiles.
2. Click New.

   The Add Index Profile panel appears:
3. Enter an **Index Profile ID**.
4. Select an index pipeline to associate with this profile.
5. Select a parser to associate with this profile.
6. Select a collection to associate with this profile.
7. Optionally, click **New params**... to enter Solr request parameters to add to the request URL when indexing documents using this profile.
8. Click **Save**.

The index profile window displays the request URL for sending documents to this profile.

**How to delete an index profile**

1. In the Fusion workspace, navigate to **Indexing > Index Profiles**.
2. In the profiles list, select the index profile to delete.
3. In the index profile configuration panel, click **Delete profile**.

**Index Pipeline Stages**

An Index Pipeline takes content and transforms it into a document suitable for indexing by Solr via a series of modular operations called stages. The objects sent from stage to stage are PipelineDocument objects. Fusion provides many specialized index stages as well as a JavaScript Index stage that allows for custom processing via a JavaScript program. The general outline of the Extract/Transform/Load processing performed by an index pipeline is:

- Raw content is parsed into one or more PipelineDocument objects.
- Any number of intermediate stages operate on the document fields directly, or, in the case of specialized NLP tools, add annotations to a document.
- Finally, the PipelineDocument is sent to Solr for indexing.

A pipeline stage definition associates a unique ID with a set of properties. Pipeline definitions are stored in ZooKeeper for reuse across pipelines and search applications. The Fusion UI provides stage-specific panels used to define and
configure each pipeline stage. Alternatively, JSON can be used to specify the sequence of pipeline stages and registered via the Fusion REST API. Some stages require additional resources, e.g., text files that contain lists of names, synonyms, places, or binary files which NLP language models. These resources can be uploaded via the Fusion UI or the REST API.

Available index pipeline stages are listed below:

**Document transformation**
- XML Transformation

**Document filtering and enrichment**
- Detect Language
- Exclude Documents
- Format Signals
- Include Documents
- JDBC Lookup
- REST Query

**Field transformation**
- Date Parsing
- Field Mapping
- Filter Short Fields
- Find and Replace
- GeoIP Lookup
- Regex Field Extraction
- Regex Field Filter
- Regex Field Replacement
- Resolve Multivalued Fields
- Solr Dynamic Field Name Mapping

**Natural language processing**
- Detect Sentences
- Gazetteer Lookup Extraction
- OpenNLP NER Extraction
- Tag Part-of-Speech

**Indexing**
- Solr Indexer
- Solr Partial Update Indexer
- Update Related Document
Troubleshooting

- Logging
- Send SMTP Email
- Send Slack Message
- Write Log Message

Advanced

- Call Pipeline
- Exclusion Filter
- Javascript
- Machine Learning
- Managed Javascript
- Set Property
- Update Experiment

Custom JavaScript Stages For Index Pipelines

The JavaScript Index stage allows you to write a custom processing logic using JavaScript to manipulate Pipeline Documents and the index pipeline context, which will be compiled by the JDK into Java bytecode that is executed by the Fusion pipeline. The first time that the pipeline is run, Fusion compiles the JavaScript program into Java bytecode using the JDK's JavaScript engine.

For a JavaScript Index stage, the JavaScript code must return either: a single document or array of documents; or the null value or an empty array. In the latter case, no further processing is possible, which means that the document will not be indexed or updated.

JavaScript Index Stage Global Variables

JavaScript is a lightweight scripting language. The JavaScript in a JavaScript stage is standard ECMAScript. What a JavaScript program can do depends on the container in which it runs. For a JavaScript Index stage, the container is a Fusion index pipeline. The following global pipeline variables are available:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc</td>
<td>PipelineDocument</td>
<td>The contents of each document submitted to the pipeline.</td>
</tr>
<tr>
<td>ctx</td>
<td>Context</td>
<td>A reference to the container that holds a map over the pipeline properties. Used to update or modify this information for downstream pipeline stages.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>collection</td>
<td>String</td>
<td>The name of the Fusion collection being indexed or queried.</td>
</tr>
<tr>
<td>solrServer</td>
<td>BufferingSolrServer</td>
<td>The Solr server instance that manages the pipeline's default Fusion collection. All indexing and query requests are done by calls to methods on this object. See SolrClient for details.</td>
</tr>
<tr>
<td>solrServerFactory</td>
<td>SolrClientFactory</td>
<td>The SolrCluster server used for lookups by collection name which returns a Solr server instance for a that collection, e.g. var productsSolr = solrServerFactory.getSolrServer(&quot;products&quot;);</td>
</tr>
</tbody>
</table>

Note

The now-deprecated global variable _context refers to the same object as ctx.

Syntax Variants

JavaScript stages can be written using **legacy syntax** or **function syntax**. The key difference between these syntax variants is how the "global variables" are used. While using legacy syntax, these variables are used as global variables. With function syntax, however, these variables are passed as function parameters.

**Legacy Syntax**

```javascript
//do some work...
return doc;
```

**Function Syntax**

```javascript
function (doc) {
  // do some work ...
  return doc;
}
```

Important

*Function syntax* is used for the examples in this document.

**JavaScript Use**

The JavaScript in a JavaScript Index stage must return either a single document or an array of documents. This can be accomplished by either:
• a series of statements where the final statement evaluates to a document or array of documents

• a function that returns a document or an array of documents

All pipeline variables referenced in the body of the JavaScript function are passed in as arguments to the function. E.g., in order to access the PipelineDocument in global variable 'doc', the JavaScript function is written as follows:

```javascript
function doWork(doc) {
    // do some work ...
    return doc;
}
```

The allowed set of function declarations are:

```javascript
function doWork(doc) { ... return doc; }
function doWork(doc, ctx) { ... return doc; }
function doWork(doc, ctx, collection) { ... return doc; }
function doWork(doc, ctx, collection, solrServer) { ... return doc; }
function doWork(doc, ctx, collection, solrServer, solrServerFactory) { ... return doc; }
```

The order of these arguments is according to the (estimated) frequency of use. The assumption is that most processing only requires access to the document object itself, and the next-most frequent type of processing requires only the document and read-only access of some context parameters. If you need to reference the solrServerFactory global variable, you must use the 5-arg function declaration.

In order to use other functions in your JavaScript program, you can define and use them, as long as the final statement in the program returns a document or documents.

**Global variable logger**

The global variable named `logger` writes messages to the log file of the server running the pipeline. This variable is truly global and doesn’t need to be declared as part of the function parameter list.

Since Fusion’s API service does the index pipeline processing, these log messages go into the log file: `fusion/5.0.x/var/log/api/api.log`. There are 5 methods available, which each take either a single argument (the string message to log) or two arguments (the string message and an exception to log). The five methods are, "debug", "info", "warn", and "error".

**JavaScript Index Stage Examples**

**Add a field to a document**

```javascript
function (doc) {
    doc.addField('some-new-field', 'some-value');
    return doc;
}
```

**Join two fields**

The following example conjoins separate latitude and longitude fields into a single geo-coordinate field, whose field name follows Solr schema conventions and ends in ".p". It also removes the original latitude and longitude fields from the document.
function(doc) {
    var value = "";
    if (doc.hasField("myGeo_Lat") && doc.hasField("myGeo_Long")) {
        value = doc.getFirstFieldValue("myGeo_Lat") + "," + doc.getFirstFieldValue("myGeo_Long");
        doc.addField("myGeo_p", value);
        doc.removeFields("myGeo_Lat");
        doc.removeFields("myGeo_Long");
        logger.debug("conjoined Lat, Long: " + value);
    }
    return doc;
}

Return an array of documents

function (doc) {
    var subjects = doc.getFieldValues("subjects");
    var id = doc.getId();
    var newDocs = [];
    for (i = 0; i < subjects.size(); i++) {
        var pd = new com.lucidworks.apollo.common.pipeline.PipelineDocument(id+'-'+i);
        pd.addField('subject', subjects.get(i));
        newDocs.push( pd );
    }
    return newDocs;
}

Parse a JSON-escaped string into a JSON object

While it's simpler to use a JSON Parsing index stage, the following code example shows you how to parse a JSON-escaped string representation into a JSON object.

This code parses a JSON object into an array of attributes, and then find the attribute "tags" which has as its value a list of strings. Each item in the list is added to a multi-valued document field named "tag_ss".

var imports = new JavaImporter(Packages.sun.org.mozilla.javascript.internal.json.JsonParser);

function(doc) {
    with (imports) {
        myData = JSON.parse(doc.getFirstFieldValue('body'));
        logger.info("parsed object");
        for (var index in myData) {
            var entity = myData[index];
            if (index == "tags") {
                for (var i=0; i<entity.length;i++) {
                    var tag = entity[i][0];
                    doc.addField("tag_ss",tag);
                }
            }
        }
    }
    doc.removeFields("body");
    return doc;
}
Do a lookup on another Fusion collection

```javascript
function doWork(doc, ctx, collection, solrServer, solrServerFactory) {
    var imports = new JavaImporter(
        org.apache.solr.client.solrj.SolrQuery,
        org.apache.solr.client.solrj.util.ClientUtils);
    with(imports) {
        var sku = doc.getFirstFieldValue("sku");
        if (!doc.hasField("mentions")) {
            var mentions = ""
            var productsSolr = solrServerFactory.getSolrServer("products");
            if (productsSolr != null) {
                var q = "sku:" + sku;
                var query = new SolrQuery();
                query.setRows(100);
                query.setQuery(q);
                var res = contactsClient.query(query);
                mentions = res.getResults().size();
                doc.addField("mentions", mentions);
            }
        }
    }
    return doc;
}
```

Reject a document

If the function returns `null` or an empty array, it will not be indexed or updated into Fusion.

```javascript
function doWork(doc) {
    if (!doc.hasField("required_field")) {
        return null;
    }
    return doc;
}
```

Debugging and Troubleshooting

To debug a JavaScript Index stage you can:

- Check the Fusion api server logs for compilation errors.
- Check the Fusion connectors server logs for runtime processing errors.
- Use the `logger` object for print debugging (in the Fusion connectors log file).
- Use the Index Workbench.

The JavaScript Engine Used by Fusion

The JavaScript engine used by Fusion is the Nashorn engine from Oracle. See The Nashorn Java API for details.

Upgrading to the latest Nashorn engine

The default version of the Nashorn engine used by Fusion versions 2.4.1 and earlier is the nashorn-0.1-jdk7.jar which contains many bugs that have since been fixed in the official JDK 1.8 version. In order to use the latest version of the
Nashorn engine, you must:

- Have an up-to-date version of Java 8 installed.
- Remove the nashorn-0.1-jdk7.jar from the Fusion classpaths:
  - cd fusion/5.0.x
  - find . -name "nashorn-0.1-jdk7.jar" -print -exec rm -i {} \\;

Creating and accessing Java types

The following information is taken from Oracle's JavaScript programming guide section 3, Using Java From Scripts.

To create script objects that access and reference Java types from Javascript use the `java.type()` function:

```javascript
var ArrayList = Java.type("java.util.ArrayList");
var a = new ArrayList;
```
Other Ingestion Methods

Usually, the simplest way to get data into Fusion is through its connectors. However, in some cases it makes sense to use other methods:

- Use a Parallel Bulk Loader (PBL) job

  Fusion Parallel Bulk Loader jobs enable bulk ingestion of structured and semi-structured data from big data systems, NoSQL databases, and common file formats like Parquet and Avro.

- Import with the REST API

  You can use the REST API to bypass the connectors and parsers and push documents directly to an index profile or index pipeline.

- Import via Pig

  You can use Pig to import data into Fusion, using the `{packageUser}-pig-functions-{connectorVersion}.jar` file found in `$FUSION_HOME/apps/connectors/resources/lucid.hadoop/jobs`.

- Import via Hive

  Fusion ships with a Serializer/Deserializer (SerDe) for Hive, included in the distribution as `{packageUser}-hive-serde-{connectorVersion}.jar` in `$FUSION_HOME/apps/connectors/resources/lucid.hadoop/jobs`.

  **Note**

  The preferred method of importing data with Hive is to use the Parallel Bulk Loader.

  - Parallel bulk loader

    This method is available with a Fusion AI license. See the Parallel Bulk Loader topic in the Fusion AI documentation.

  - Batch ingestion of signals is also available with a Fusion AI license.

Import Data with the Parallel Bulk Loader

Fusion Parallel Bulk Loader jobs enable bulk ingestion of structured and semi-structured data from big data systems, NoSQL databases, and common file formats like Parquet and Avro.

The Parallel Bulk Loader leverages the popularity of Spark as a prominent distributed computing platform for big data. A number of companies invest heavily in building production-ready Spark SQL data source implementations for big data and NoSQL systems, much as Lucidworks has done for Solr. The Parallel Bulk Loader uses connectors provided by the experts who develop these complex systems.

This feature is available with a Fusion AI license. See the Fusion AI User Guide for details.

Import Data with Hive

Fusion ships with a Serializer/Deserializer (SerDe) for Hive, included in the distribution as `lucidworks-hive-serde-2.2.6.jar` in `$FUSION_HOME/apps/connectors/resources/lucid.hadoop/jobs`. 
The preferred method of importing data with Hive is to use the Parallel Bulk Loader.

### Features

- Index Hive table data to Solr.
- Read Solr index data to a Hive table.
- Kerberos support for securing communication between Hive and Solr.
- As of v2.2.4 of the SerDe, integration with Lucidworks Fusion is supported.
  - Fusion’s index pipelines can be used to index data to Fusion.
  - Fusion’s query pipelines can be used to query Fusion’s Solr instance for data to insert into a Hive table.

### Add the SerDe Jar to Hive Classpath

In order for the Hive SerDe to work with Solr, the SerDe jar must be added to Hive’s classpath using the `hive.aux.jars.path` capability. There are several options for this, described below.

It’s considered a best practice to use a single directory for all auxiliary jars you may want to add to Hive so you only need to define a single path. However, you must then copy any jars you want to use to that path.

1. If you use Hive with Ambari (as with the Hortonworks HDP distribution), go to menu:Hive[Configs > Advanced], and scroll down to menu:Advanced hive-env[hive-env template]. Find the section where the `HIVE_AUX_JARS_PATH` is defined, and add the path to each line which starts with `export`. What you want will end up looking like:

   ```bash
   # Folder containing extra libraries required for hive compilation/execution can be controlled by:
   if [ "${HIVE_AUX_JARS_PATH}" != "" ]; then
     if [ -f "${HIVE_AUX_JARS_PATH}" ]; then
       export HIVE_AUX_JARS_PATH=${HIVE_AUX_JARS_PATH},/usr/hive/auxlib
     else
       export HIVE_AUX_JARS_PATH=usr/hdp/current/hive-webhcat/share/hcatalog/hive-hcatalog-core.jar,usr/hive/auxlib
     fi
   else
     export HIVE_AUX_JARS_PATH=usr/hdp/current/hive-webhcat/share/hcatalog/hive-hcatalog-core.jar,usr/hive/auxlib
   fi
   ```

2. If not using Ambari or similar cluster management tool, you can add the jar location to `hive/conf/hive-site.xml`:

   ```xml
   <property>
     <name>hive.aux.jars.path</name>
     <value>/usr/hive/auxlib</value>
   </property>
   ```

3. Another option is to launch Hive with the path defined with the `auxpath` variable:
There are also other approaches that could be used. Keep in mind, though, that the jar **must** be loaded into the classpath, adding it with the **ADD JAR** function is not sufficient.

**Indexing Data to Fusion**

If you use Lucidworks Fusion, you can index data from Hive to Solr via Fusion’s index pipelines. These pipelines allow you several options for further transforming your data.

| Tip | If you are using Fusion v3.0.x, you already have the Hive SerDe in Fusion’s ../apps/connectors/resources/lucid.hadoop/jobs directory.
The SerDe jar that supports Fusion is v2.2.4 or higher. This was released with Fusion 3.0.
If you are using Fusion 3.1.x and higher, you will need to download the Hive SerDe from [http://lucidworks.com/ connectors/](http://lucidworks.com/connectors/). Choose the proper Hadoop distribution and the resulting .zip file will include the Hive SerDe.
A 2.2.4 or higher jar built from this repository will also work with Fusion 2.4.x releases. |
|---|---|

This is an example Hive command to create an external table to index documents in Fusion and to query the table later.

```
hive> CREATE EXTERNAL TABLE fusion (id string, field1_s string, field2_i int)
    STORED BY 'com.lucidworks.hadoop.hive.FusionStorageHandler'
    LOCATION '/tmp/fusion'
    TBLPROPERTIES('fusion.endpoints' = 'http://localhost:8764/api/apollo/index-pipelines/<pipeline>/collections/<collection>/index',
    'fusion.fail.on.error' = 'false',
    'fusion.buffer.timeouts' = '1000',
    'fusion.batchSize' = '500',
    'fusion.realm' = 'KERBEROS',
    'fusion.user' = 'fusion-indexer@FUSIONSERVER.COM',
    'java.security.auth.login.config' = '/path/to/JAAS/file',
    'fusion.jaas.appname' = 'FusionClient',
    'fusion.query' = '*:*');
```

In this example, we have created an external table named “fusion”, and defined a custom storage handler (**STORED BY** 'com.lucidworks.hadoop.hive.FusionStorageHandler') that a class included with the Hive SerDe jar designed for use with Fusion.

Note that all of the same caveats about field types discussed in the section [Defining Fields for Solr](#) apply to Fusion as well. In Fusion, however, you have the option of using an index pipeline to perform specific field mapping instead of using dynamic fields.
The LOCATION indicates the location in HDFS where the table data will be stored. In this example, we have chosen to use /tmp/fusion.

In the section TBLPROPERTIES, we define several properties for Fusion so the data can be indexed to the right Fusion installation and collection:

**fusion.endpoints**

The full URL to the index pipeline in Fusion. The URL should include the pipeline name and the collection data will be indexed to.

**fusion.fail.on.error**

If true, when an error is encountered, such as if a row could not be parsed, indexing will stop. This is false by default.

**fusion.buffer.timeoutms**

The amount of time, in milliseconds, to buffer documents before sending them to Fusion. The default is 1000. Documents will be sent to Fusion when either this value or fusion.batchSize is met.

**fusion.batchSize**

The number of documents to batch before sending the batch to Fusion. The default is 500. Documents will be sent to Fusion when either this value or fusion.buffer.timeoutms is met.

**fusion.realm**

This is used with fusion.user and fusion.password to authenticate to Fusion for indexing data. Two options are supported, KERBEROS or NATIVE. Kerberos authentication is supported with the additional definition of a JAAS file. The properties java.security.auth.login.config and fusion.jaas.appname are used to define the location of the JAAS file and the section of the file to use. Native authentication uses a Fusion-defined username and password. This user must exist in Fusion, and have the proper permissions to index documents.

**fusion.user**

The Fusion username or Kerberos principal to use for authentication to Fusion. If a Fusion username is used ('fusion.realm' = 'NATIVE'), the fusion.password must also be supplied.

**fusion.password**

This property is not shown in the example above. The password for the fusion.user when the fusion.realm is NATIVE.

**java.security.auth.login.config**

This property defines the path to a JAAS file that contains a service principal and keytab location for a user who is authorized to read from and write to Fusion and Hive. The JAAS configuration file must be copied to the same path on every node where a Node Manager is running (i.e., every node where map/reduce tasks are executed). Here is a sample section of a JAAS file:

```java
Client { (1)
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/data/fusion-indexer.keytab" (2)
    storeKey=true
    useTicketCache=true
    debug=true
    principal="fusion-indexer@FUSIONSERVER.COM"; (3)
};
```
Query and Insert Data to Hive

Once the table is configured, any syntactically correct Hive query will be able to query the index.

For example, to select three fields named "id", "field1_s", and "field2_i" from the "solr" table, you would use a query such as:

```
hive> SELECT id, field1_s, field2_i FROM solr;
```

Replace the table name as appropriate to use this example with your data.

To join data from tables, you can make a request such as:

```
hive> SELECT id, field1_s, field2_i FROM solr left JOIN sometable right WHERE left.id = right.id;
```

And finally, to insert data to a table, simply use the Solr table as the target for the Hive INSERT statement, such as:

```
hive> INSERT INTO solr
    SELECT id, field1_s, field2_i FROM sometable;
```

Example Indexing Hive to Solr

Solr includes a small number of sample documents for use when getting started. One of these is a CSV file containing book metadata. This file is found in your Solr installation, at `$SOLR_HOME/example/exampledocs/books.csv`.

Using the sample `books.csv` file, we can see a detailed example of creating a table, loading data to it, and indexing that data to Solr.
CREATE TABLE books (id STRING, cat STRING, title STRING, price FLOAT, in_stock BOOLEAN, author STRING, series STRING, seq INT, genre STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','; (1)

LOAD DATA LOCAL INPATH '/solr/example/examledocs/books.csv' OVERWRITE INTO TABLE books; (2)

CREATE EXTERNAL TABLE solr (id STRING, cat_s STRING, title_s STRING, price_f FLOAT, in_stock_b BOOLEAN, author_s STRING, series_s STRING, seq_i INT, genre_s STRING) STORED BY 'com.lucidworks.hadoop.hive.LWStorageHandler' (4)
 LOCATION '/tmp/solr' (5)
 TBLPROPERTIES('solr.zkhost' = 'zknode1:2181,zknode2:2181,zknode3:2181/solr',
 'solr.collection' = 'gettingstarted',
 'solr.query' = '*:*', (6)
 'lww.jaas.file' = '/data/jaas-client.conf'); (7)

INSERT OVERWRITE TABLE solr SELECT b.* FROM books b;

1. Define the table books, and provide the field names and field types that will make up the table.
2. Load the data from the books.csv file.
3. Create an external table named solr, and provide the field names and field types that will make up the table. These will be the same field names as in your local Hive table, so we can index all of the same data to Solr.
4. Define the custom storage handler provided by the lucidworks-hive-serde-2.2.6.jar.
5. Define storage location in HDFS.
6. The query to run in Solr to read records from Solr for use in Hive.
7. Define the location of Solr (or ZooKeeper if using SolrCloud), the collection in Solr to index the data to, and the query to use when reading the table. This example also refers to a JAAS configuration file that will be used to authenticate to the Kerberized Solr cluster.

Import Data with Pig

You can use Pig to import data into Fusion, using the lucidworks-pig-functions-2.2.6.jar file found in $FUSION_HOME/apps/connectors/resources/lucid.hadoop/jobs.

Available Functions

The Pig functions included in the lucidworks-pig-functions-2.2.6.jar are three UserDefined Functions (UDF) and two Store functions. These functions are:

- com/lucidworks/hadoop/pig/SolrStoreFunc.class
- com/lucidworks/hadoop/pig/FusionIndexPipelinesStoreFunc.class
- com/lucidworks/hadoop/pig/EpochToCalendar.class
- com/lucidworks/hadoop/pig/Extract.class
- com/lucidworks/hadoop/pig/Histogram.class

Using The Functions

Register the Functions

There are two approaches to using functions in Pig: REGISTER them in the script, or load them with your Pig command.
If using REGISTER, the Pig function jars must be put in HDFS in order to be used by your Pig script. It can be located anywhere in HDFS; you can either supply the path in your script or use a variable and define the variable with -p property definition.

The example below uses the second approach, loading the jars with the -Dpig.additional.jars system property when launching the script. With this approach, the jars can be located anywhere on the machine where the script will be run.

### Indexing Data to Fusion

When indexing data to Fusion, there are several parameters to pass with your script in order to output data to Fusion for indexing.

These parameters can be made into variables in the script, with the proper values passed on the command line when the script is initiated. The example script below shows how to do this for Solr. The theory is the same for Fusion, only the parameter names would change as appropriate:

**fusion.endpoints**

The full URL to the index pipeline in Fusion. The URL should include the pipeline name and the collection data will be indexed to.

**fusion.fail.on.error**

If true, when an error is encountered, such as if a row could not be parsed, indexing will stop. This is false by default.

**fusion.buffer.timeoutms**

The amount of time, in milliseconds, to buffer documents before sending them to Fusion. The default is 1000. Documents will be sent to Fusion when either this value or fusion.batchSize is met.

**fusion.batchSize**

The number of documents to batch before sending the batch to Fusion. The default is 500. Documents will be sent to Fusion when either this value or fusion.buffer.timeoutms is met.

**fusion.realm**

This is used with fusion.user and fusion.password to authenticate to Fusion for indexing data. Two options are supported, KERBEROS or NATIVE. * Kerberos authentication is supported with the additional definition of a JAAS file. The properties java.security.auth.login.config and fusion.jaas.appname are used to define the location of the JAAS file and the section of the file to use. These are described in more detail below. * Native authentication uses a Fusion-defined username and password. This user must exist in Fusion, and have the proper permissions to index documents.

**fusion.user**

The Fusion username or Kerberos principal to use for authentication to Fusion. * If a Fusion username is used ('fusion.realm' = 'NATIVE'), the fusion.password must also be supplied.

**fusion.pass**

This property is not shown in the example above. The password for the fusion.user when the fusion.realm is NATIVE.

### Indexing to a Kerberized Fusion Installation

When Fusion is secured with Kerberos, Pig scripts must include the full path to a JAAS file that includes the service principal and the path to a keytab file that will be used to index the output of the script to Fusion.
Additionally, a Kerberos ticket must be obtained on the server for the principal using `kinit`.

`java.security.auth.login.config`

This property defines the path to a JAAS file that contains a service principal and keytab location for a user who is authorized to write to Fusion. The JAAS configuration file **must** be copied to the same path on every node where a Node Manager is running (i.e., every node where map/reduce tasks are executed). Here is a sample section of a JAAS file:

```java
Client { (1)
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/data/fusion-indexer.keytab" (2)
    storeKey=true
    useTicketCache=true
    debug=true
    principal="fusion-indexer@FUSIONSERVER.COM"; (3)
};
```

1. The name of this section of the JAAS file. This name will be used with the `fusion.jaas.appname` parameter.
2. The location of the keytab file.
3. The service principal name. This should be a different principal than the one used for Fusion, but must have access to both Fusion and Pig. This name is used with the `fusion.user` parameter described above.

`fusion.jaas.appname`:
Used only when indexing to or reading from Fusion when it is secured with Kerberos.

This property provides the name of the section in the JAAS file that includes the correct service principal and keytab path.

**Sample CSV Script**

The following Pig script will take a simple CSV file and index it to Solr.

```pig
set solr.zkhost '$zkHost';
set solr.collection '$collection'; (1)

A = load '$csv' using PigStorage(',\') as
(id_s:chararray,city_s:chararray,country_s:chararray,code_s:chararray,code2_s:chararray,latitude_s:chararray,longtitude_s:chararray,flag_s:chararray); (2)
--dump A;
B = FOREACH A GENERATE $0 as id, 'city_s', $1, 'country_s', $2, 'code_s', $3, 'code2_s', $4, 'latitude_s', $5, 'longitude_s', $6, 'flag_s', $7; (3)
ok = store B into 'SOLR' using com.lucidworks.hadoop.pig.SolrStoreFunc(); (4)
```

This relatively simple script is doing several things that help to understand how the Solr Pig functions work.

1. This and the line above define parameters that are needed by `SolrStoreFunc` to know where Solr is. `SolrStoreFunc` needs the properties `solr.zkhost` and `solr.collection`, and these lines are mapping the `zkhost` and `collection` parameters we will pass when invoking Pig to the required properties.

2. Load the CSV file, the path and name we will pass with the `csv` parameter. We also define the field names for each
column in CSV file, and their types.

3. For each item in the CSV file, generate a document id from the first field ($0) and then define each field name and value in name, value pairs.

4. Load the documents into Solr, using the SolrStoreFunc. While we don't need to define the location of Solr here, the function will use the zkhost and collection properties that we will pass when we invoke our Pig script.

| Warning | When using SolrStoreFunc, the document ID must be the first field. |

When we want to run this script, we invoke Pig and define several parameters we have referenced in the script with the -p option, such as in this command:

```
./bin/pig -Dpig.additional.jars=/path/to/lucidworks-pig-functions-2.2.6.jar -p 
csv=/path/to/my/csv/airports.dat -p zkHost=zknode1:2181,zknode2:2181,zknode3:2181/solr -p 
collection=myCollection ~/myScripts/index-csv.pig
```

The parameters to pass are:

| csv | The path and name of the CSV file we want to process. |
| zkhost | The ZooKeeper connection string for a SolrCloud cluster, in the form of zkhost1:port,zkhost2:port,zkhost3:port/chroot. In the script, we mapped this to the solr.zkhost property, which is required by the SolrStoreFunc to know where to send the output documents. |
| collection | The Solr collection to index into. In the script, we mapped this to the solr.collection property, which is required by the SolrStoreFunc to know the Solr collection the documents should be indexed to. |

| Tip | The zkhost parameter above is only used if you are indexing to a SolrCloud cluster, which uses ZooKeeper to route indexing and query requests. |

If, however, you are not using SolrCloud, you can use the solrUrl parameter, which takes the location of a standalone Solr instance, in the form of http://host:port/solr.

In the script, you would change the line that maps solr.zkhost to the zkhost property to map solr.server.url to the solrUrl property. For example:

```
`set solr.server.url '$solrUrl';`
```
Import Data with the REST API

It is often possible to get documents into Fusion Server by configuring a datasource with the appropriate connector. (See the Connectors documentation for details.)

But if there are obstacles to using connectors, it can be simpler to index documents with a REST API call to an index profile or pipeline.

Push documents to Fusion using index profiles

Index profiles allow you to send documents to a consistent endpoint (the profile alias) and change the backend index pipeline as needed. The profile is also a simple way to use one pipeline for multiple collections without any one collection "owning" the pipeline. See Index Profiles.

You can send documents directly to an index using the Index REST API. The request path is:

/api/apps/<app_name>/index/<index-profile-id>

These requests are sent as a POST request. The request header specifies the format of the contents of the request body. Create an index profile in the Fusion UI.

To send a streaming list of JSON documents, you can send the JSON file that holds these objects to the API listed above with application/json as the content type. If your JSON file is a list or array of many items, the endpoint operates in a streaming way and indexes the docs as necessary.

Send data to an index profile that is part of an app

Accessing an index profile through an app lets a Fusion admin secure and manage all objects on a per-app basis. Security is then determined by whether a user can access an app. This is the recommended way to manage permissions in Fusion.

The syntax for sending documents to an index profile that is part of an app is as follows:

```bash
curl -u <username>:<password> -X POST -H 'content-type: application/json'
http://localhost:8764/api/apps/<my_app>/index/<index-profile-id> --data-binary @my-json-data.json
```

Note that spaces in an app name become underscores, while spaces in an index profile name become hyphens.

To prevent the terminal from displaying all the data and metadata it indexes, for example if you are indexing a large file, you can optionally append ?echo=false to the URL.

Be sure to set the content type header properly for the content being sent. Some frequently used content types are:

- Text: application/json, application/xml
- PDF documents: application/pdf
- MS Office:
  - .docx: application/vnd.openxmlformats-officedocument.wordprocessingml.document
  - .xlsx: application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
  - .pptx: application/vnd.openxmlformats-officedocument.presentationml.presentation
Example: Send JSON data to an index profile under an app

In $FUSION_HOME/apps/solr-dist/example/exampledocs you can find a few sample documents. This example uses one of these, books.json.

To push JSON data to an index profile under an app:

1. Create an index profile. In the Fusion UI, click **Indexing > Index Profiles** and follow the prompts.

   In this example, the index profile is named **test index profile**, and it resides in the Fusion app named **test app**.

2. From the directory containing books.json, enter the following, substituting your values for username, password, app name, and index profile name:

   ```
curl -u <username>:<password> -X POST -H 'content-type: application/json'  
   http://localhost:8764/api/apps/<test_app>/index/<test-index-profile>  
   --data-binary @books.json
   ```

3. Test that your data has made it into Fusion:

   a. Log into the Fusion UI.
   b. Navigate to the app where you sent your data.
   c. Navigate to the Query Workbench.
   d. Search for :.
   e. Select relevant Display Fields, for example **author** and **name**.

Example: Send JSON data without defining an app

In most cases it is best to delegate permissions on a per-app basis. But if your use case requires it, you can push data to Fusion without defining an app.

To send JSON data without app security, issue the following curl command:

```
curl -u <username>:<password> -X POST -H 'content-type: application/json'  
   http://localhost:8764/api/index/<index-profile-id>  
   --data-binary @my-json-data.json
   ```

Example: Send XML data to an index profile with an app

To send XML data to an app, use the following:

```
curl -u <username>:<password> -X POST -H 'content-type: application/xml'  
   http://localhost:8764/api/apps/<test_app>/index/<index-profile-id>  
   --data-binary @my-xml-file.xml
   ```

Documents can be created on the fly using the PipelineDocument JSON notation.

Send documents to an index pipeline

Although sending documents to an index profile is recommended, if your use case requires it, you can send documents
directly to an index pipeline.

See the index pipeline REST API reference documentation at Index Pipelines API.

**Specify a parser**

When you push data to a pipeline, you can specify the name of the parser by adding a parserId querystring parameter to the URL. For example: `http://myserver/api/index-pipelines/mypipe/collections/mycoll/index?parserId=myparser`

If you do not specify a parser, and you are indexing outside of an app (`http://myserver/api/index-pipelines/…`), then the _system parser is used.

If you do not specify a parser, and you are indexing in an app context (`http://myserver/api/apps/myapp/index-pipelines/…`), then the parser with the same name as the app is used.

**Send a PDF document to an index pipeline**

Index a PDF document through the `conn_solr` index pipeline to a collection named `docs`. The pre-configured `conn_solr` pipeline includes stages to parse documents with Tika, map fields, and index the documents to Solr (in that order).

```
```

**Indexing CSV Files**

In the usual case, to index a CSV (or TSV) file, the file is split into records, one per row, and each row is indexed as a separate document.
Blob Storage

Fusion accepts large binary objects (blobs) for upload, and stores them in Solr. Blob uploads are used to install models, lookup lists, JDBC drivers, connectors, and more.

Blob Types

A resourceType query parameter can be used to specify the a blob type. For example, specify plugin:connector when uploading a connector, like this:

curl -H 'content-type:application/zip' -X PUT 'fusion-host:{api-port}/api/blobs/myplugin?resourceType=plugin:connector' --data-binary @myplugin.zip

The complete list of valid values for resourceType is below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>banana</td>
<td>A Banana dashboard</td>
</tr>
<tr>
<td>catalog</td>
<td>An analytics catalog</td>
</tr>
<tr>
<td>driver:jdbc</td>
<td>A JDBC driver</td>
</tr>
<tr>
<td>file:js-index</td>
<td>A JavaScript file for use with a Managed Javascript index stage.</td>
</tr>
<tr>
<td>file:js-query</td>
<td>A JavaScript file for use with a Managed Javascript query stage.</td>
</tr>
<tr>
<td>file-upload</td>
<td>Any uploaded file, such as from the Quickstart or the Index Workbench.</td>
</tr>
<tr>
<td>model:ml-model</td>
<td>A machine learning model (Fusion AI only)</td>
</tr>
<tr>
<td>model:open-nlp</td>
<td>An OpenNLP model (Fusion AI only)</td>
</tr>
<tr>
<td>other</td>
<td>A blob of unknown type</td>
</tr>
<tr>
<td>plugin:connector</td>
<td>A connector plugin</td>
</tr>
</tbody>
</table>

If no resourceType is specified on upload, "other" is assigned by default.
The Blob manager

In addition to the Blob Store API, the Fusion UI provides an interface to the blob store in the Fusion workspace at System > Blobs:
• Click **Add** to upload a new blob:
Note

In Fusion 4.1, it is not possible to upload a blob of type `file:js-index` or `file:js-query` using the Fusion UI. Instead, use the Blob Store API. These blob types become visible in the blob manager after upload.

- Select an uploaded blob to view, replace, or delete it:
Getting Data Out

- Query pipelines process search requests before sending them to Fusion Server’s Solr core. A query pipeline can perform transformations on a search request in order to customize the search results that Solr returns. See Query Pipeline Configuration.

- Search applications are the front-end interfaces that you build on top of Fusion. Your application makes calls to Fusion's REST API in order to retrieve search results or perform other actions. Certain features, like autocomplete and synonyms, require some configuration on the Fusion back end. See Application Development.

- Whenever you’re getting data out of Fusion Server, you may find it handy to consult the Query Language Cheat Sheet.

If you are using Fusion AI with Fusion Server, you can generate and retrieve additional data for analysis or to enhance the end-user experience. For example, Fusion AI can produce sophisticated recommendations to guide end users to the best available results, including results that don’t exactly match the original user-submitted query. It can also perform machine learning functions that automatically improve search results based on the past activities of users. See the Fusion AI documentation.
Query Pipeline Configuration

- Query pipelines filter, transform, and augment Solr queries and responses in order to return all and only the most relevant search results.
- Query pipeline stages are the components of a query pipeline, configured and ordered to produce the desired search results.
- Query profiles provide aliases for query pipelines, enabling your application to point to the same endpoint regardless of the query pipeline behind it. A query profile can also specify the collection and/or configuration parameters for the query pipeline, if abstraction is desired for those aspects.
- Custom Javascript query stages are a versatile way to handle a wide range of special requirements in your pipeline.
- The stored parameters collection stores Solr facet parameters for a particular document or product category.

Query Pipelines

A Query Pipeline transforms a set of inputs into a Solr query request and it can execute requests and manipulate the Solr response as well, via a set of modularized operations called Query Stages. The objects sent from stage to stage are Request objects and Response objects.

Fusion stores pipeline names and definitions, allowing a pipeline to be reused across applications. Pipeline definitions can be modified, so that as an application evolves, the pipelines used by that application can evolve accordingly. During application development, the Fusion UI can be used to develop and debug a Query Pipeline.

The available stage types allow setting specific parameters for the query, such as the number of results to return or the query parser to use. You can also define facets and recommendations to be returned with the results. If Access Control Lists (ACLs) are in use, you can apply a security-trimming stage to apply user access restrictions to the results.

For details about the available REST APIs, see Query Pipelines API and Query Stages API.

Default Query Pipelines

When you create a new app with a default collection, the collection includes a default query pipeline. When you create a new collection in an existing app, Fusion also creates a default query pipeline for the new collection. The pipeline name is the same as the collection name.

The default query pipeline has the following pre-configured stages:

- Text Tagger
  
  This stage uses the SolrTextTagger handler to identify known entities in the query by searching the _query_rewrite collection (or the _query_rewrite_staging collection in the case of the Fusion AI query rewriting Simulator) to find matching spelling corrections, phrase boosts, underperforming query improvements, and synonym expansions in order to perform query rewriting.

- Boost with Signals
  
  The Boost with Signals query pipeline stage uses aggregated signals to selectively boost items in the set of search results.

- Query Fields
The Query Fields query pipeline stage defines common Solr query parameters for the edismax query parser. An alternative to this stage is the Additional Query Parameters stage.

- Field Facet

The Field Facet query pipeline stage is used to add a Solr Field Facet query to the search query pipeline.

- Apply Rules
- A Solr Query

The Solr Query stage transforms the Fusion query pipeline Request object into a Solr query and sends it to Solr.

- Modify Response with Rules

Most rules operate on the request, but some rule types, such as banner rules or redirect rules, do their work when the response comes back. The Modify Response with Rules stage applies those rules to the response. For example, a banner rule can add a banner URL to the response before returning it to the client.

Custom Query Pipelines

Using the Query Workbench or the REST API, you can develop custom pipelines to suit any search application. Start with any of Fusion's built-in query pipelines, then add, remove, and re-order the pipeline stages as needed to produce the appropriate query results.

Asynchronous query pipeline processing

Query pipeline processing performance can be improved by enabling asynchronous processing for certain stages that make requests to secondary collections, external databases, and so on. The following stages support asynchronous processing:

- Active Directory Security Trimming
- Apply Rules
- Boost with Signals
- JDBC Lookup
- Security Trimming
- Solr Subquery

This feature uses the fork-and-join model, where any of the stages above can create a fork on the pipeline. The parallel processes are joined again using the Merge Async Results stage at a later point in the pipeline.

How to enable asynchronous query pipeline processing

1. In the Query Workbench, open the query pipeline you want to modify.
2. Click Add a Stage and select a stage that supports asynchronous processing to your pipeline, or select the stage in your pipeline if it has already been added.
   
   Stages that support this are listed above.

3. In the stage configuration panel, select Asynchronous Execution Config.
4. Select **Enable Async Execution**.

5. Copy the **Async ID** value.

6. Click **Apply**.

   Verify that the stage is in the correct position in your pipeline.

7. Click **Add a Stage** and select the Merge Async Results stage.
8. In the stage configuration panel, click the Add button next to Async IDs.

9. Paste the ID from the asynchronous query stage into the new ID field.

**Tip**

When using multiple asynchronous stages, the order in which IDs are entered here is the order in which they are joined back into the pipeline.

10. Click Apply.

The Merge Async Results stage is now the first stage in the pipeline.

11. Drag the Merge Async Results stage down so that it appears immediately before the Solr Query stage:
12. Click **Save**.

**Query Pipeline Stages**

A query pipeline is made up of a series of query stages that process incoming search queries.

A pipeline stage definition associates a unique ID with a set of properties. These definitions are registered with the Fusion API service and stored in ZooKeeper for re-use across pipelines and search applications.

Fusion includes a number of specialized query stages as well as a JavaScript stage that allows advanced processing via a JavaScript program.

**Configuring query pipeline stages**

- In the Fusion UI, the Query Workbench provides an environment for configuring the stages in a query pipeline.

- The Query Stages API is used to create, list, update, or delete query stages using JSON. See also the Query Pipelines API.

**Asynchronous query pipeline processing**

Query pipeline processing performance can be improved by enabling asynchronous processing for certain stages that make requests to secondary collections, external databases, and so on. The following stages support asynchronous processing:

- Active Directory Security Trimming

- Apply Rules

- Boost with Signals
This feature uses the fork-and-join model, where any of the stages above can create a fork on the pipeline. The parallel processes are joined again using the Merge Async Results stage at a later point in the pipeline.

How to enable asynchronous query pipeline processing

1. In the Query Workbench, open the query pipeline you want to modify.
2. Click **Add a Stage** and select a stage that supports asynchronous processing to your pipeline, or select the stage in your pipeline if it has already been added.

   Stages that support this are listed above.

3. In the stage configuration panel, select **Asynchronous Execution Config**.
4. Select **Enable Async Execution**.

   ![Asynchronous Execution Config](image)

   | Tip | Fusion automatically assigns an **Async ID** value to this stage. Change this to a more memorable string that describes the asynchronous stages you are merging, such as **signals** or **access_control**.

5. Copy the **Async ID** value.
6. Click **Apply**.

   Verify that the stage is in the correct position in your pipeline.
7. Click **Add a Stage** and select the Merge Async Results stage.

8. In the stage configuration panel, click the Add button next to **Async IDs**.

9. Paste the ID from the asynchronous query stage into the new ID field.

   **Tip**

   When using multiple asynchronous stages, the order in which IDs are entered here is the order in which they are joined back into the pipeline.

10. Click **Apply**.

    The Merge Async Results stage is now the first stage in the pipeline.

11. Drag the Merge Async Results stage down so that it appears immediately before the Solr Query stage:
12. Click **Save**.

**Conditional query processing**

Query Pipeline stages are used to modify Request objects and Response objects. Each stage can include a conditional JavaScript expression (the **condition** property in its configuration) that can access these objects.

For example, this condition first checks that the property **fusion-user-name** is specified in the Request object, then checks for a particular value:

```
request.hasParam("fusion-user-name") && request.getFirstParam("fusion-user-name").equals("SuperUser");
```

**Reference topics**

See these reference topics for complete details about each query pipeline stage:

**Setup**

- Active Directory Security Trimming
- Field Facet
- Query Fields
- Security Trimming

**Relevancy**

- Block Documents
• Landing Pages
• Parameterized Boosting
• Recommend More Like This
• Recommend Items for User
• Recommend Items for Item

Query rewriting

• Text Tagger
• Apply Rules

Response rewriting

• Machine Learning (Responses) Stage
• Response Document Exclusion Stage
• Response Document Field Redaction Stage
• Response Pairwise Swap
• Response Shuffle Stage
• Modify Response with Rules

Fetch data

• JDBC Lookup
• REST Query
• Solr Query
• Solr Subquery

Troubleshooting

• Logging
• Send Slack Message
• Send SMTP Email
• Write Log Message

Advanced

• Additional Query Parameters
• Javascript
• Managed Javascript
• Retrieve Stored Parameters
Query Profiles

Query profiles let you consistently point your search application at a static endpoint, but give you the flexibility to change the actual query pipeline being used (and optionally, the collection and/or configuration parameters for the query pipeline).

For example, an e-commerce site might want to create a query pipeline to support a month-long promotion. After the query pipeline is configured, it can be easily enabled by changing the query profile in use by the front-end application to use the new pipeline.

To run a query through a query profile, use the REST API as described below. For CRUD operations, you can use either the REST API or the Fusion UI.

Rules Simulator query profile

The Rules Simulator allows product owners to experiment with rules and other query rewrites in the _query_rewrite_staging collection before deploying them to the _query_rewrite collection.

Each app has a _rules_simulator query profile, configured to use the _query_rewrite_staging collection for query rewrites instead of the _query_rewrite collection. This profile is created automatically whenever a new app is created.

Query profiles in the REST API

- Query Profiles API (/query-profiles)
  
  Create, read, update, and delete query profiles.

- Query API (/query)
  
  Run a query through a query profile by specifying the profile ID and appending the request with a Solr query string, as in /api/query/<id>?<solrQuery>.

Query profiles in the UI

Query profiles are configured at Querying > Query Profiles.
How to create a new query profile

1. In the Fusion workspace, navigate to Querying > Query Profiles.
2. Click New.

The Add Query Profile panel appears:
3. Enter an **Query Profile ID**.

4. Select a query pipeline to associate with this profile.

5. Enter a **Solr search handler** to use with this profile.

   The default of `select` is usually fine.

6. Select a collection to associate with this profile.

7. Optionally, click **New params...** to enter Solr request parameters to add to the request URL when submitting queries using this profile.

8. Optionally, select **Enable experimentation** to configure this profile for use with experiments.
   a. Select the percentage of experiment traffic to direct to this profile.
   b. Select the experiment to associate with this profile, or select **Add Experiment** to configure a new one.

   See Set up an experiment for details.

9. Click **Save**.

   The query profile window displays the request URL for sending queries to this profile.

---

**How to delete a query profile**

1. In the Fusion workspace, navigate to **Querying > Query Profiles**.

2. In the profiles list, select the query profile to delete.

3. In the query profile configuration panel, click **Delete profile**.

---

**Custom JavaScript Stages for Query Pipelines**

The JavaScript Query stage allows you to write a custom processing logic using JavaScript to manipulate search requests and responses. The first time that the pipeline is run, Fusion compiles the JavaScript program into Java bytecode using the JDK's JavaScript engine.

The JavaScript Query stage allows you to run JavaScript functions over search requests and responses by manipulating variables called "request" and "response" which are Request objects and Response objects, respectively.

**JavaScript Query Stage Global Variables**

JavaScript is a lightweight scripting language. The JavaScript in a JavaScript stage is standard ECMAScript. What a JavaScript program can do depends on the container in which it runs. For a JavaScript Query stage, the container is a Fusion query pipeline. The following global pipeline variables are available:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Request</td>
<td>The Solr query information.</td>
</tr>
<tr>
<td>response</td>
<td>Response</td>
<td>The Solr response information.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ctx</td>
<td>Context</td>
<td>A reference to the container which holds a map over the pipeline properties. Used to update or modify this information for downstream pipeline stages.</td>
</tr>
<tr>
<td>collection</td>
<td>String</td>
<td>The name of the Fusion collection being indexed or queried.</td>
</tr>
<tr>
<td>solrServer</td>
<td>BufferingSolrServer</td>
<td>The Solr server instance that manages the pipeline’s default Fusion collection. All indexing and query requests are done by calls to methods on this object. See SolrClient for details.</td>
</tr>
<tr>
<td>solrServerFactory</td>
<td>SolrClientFactory</td>
<td>The SolrCluster server used for lookups by collection name which returns a Solr server instance for that collection, e.g. var productsSolr = solrServerFactory.getSolrServer(&quot;products&quot;);</td>
</tr>
</tbody>
</table>

**Note**

The now-deprecated global variable `_context` refers to the same object as `ctx`.

**Syntax Variants**

JavaScript stages can be written using **legacy syntax** or **function syntax**. The key difference between these syntax variants is how the "global variables" are used and interpreted. While using legacy syntax, these variables are used as global variables. With function syntax, however, these variables are passed as function parameters.

**Legacy Syntax**

```javascript
request.addParam("foo", "bar");
```

**Function Syntax**

```javascript
function(request,response) {
    request.addParam("foo", "bar");
}
```

**Important**

*Function syntax* is used for the examples in this document.
Global variable logger

The global variable named logger writes messages to the log file of the server running the pipeline. This variable is truly global and doesn't need to be declared as part of the function parameter list.

Since Fusion’s API service does the query pipeline processing, these log messages go into the log file: fusion/5.0.x/var/log/api/api.log. There are 5 methods available, which each take either a single argument (the string message to log) or two arguments (the string message and an exception to log). The five methods are, "debug", "info", "warn", and "error".

JavaScript Query Stage Examples

Add a parameter to the query request

```javascript
function(request,response , ctx, collection, solrServer, solrServerFactory) {
    request.addParam("foo", "bar");
}
```

Add a parameter to the query response

This example contains a simple JavaScript function which copies information from the ctx (Context) object into the query response. Requirements:

- The response parameter cannot be used in query pipeline stages prior to the Solr query stage. This example assumes use in a later stage.
- The response.initialEntity.appendStringList() function only works if the query's wt (writer type) parameter is set to json or xml, for example: wt=json.

```javascript
(function () {
    "use strict";

    var List = Java.type('java.util.List');

    function add_to_response(key, list) {
        if (list.length > 0) {
            response.initialEntity.appendStringList(key, Java.to(list, List));
        }
    }

    return function(request,response , ctx, collection, solrServer, solrServerFactory) {
        add_to_response('banners', ctx.getProperty('banners'));
        add_to_response('landing-pages', ctx.getProperty('redirects'));
    };
})();
```

Manually Adding Dependencies

To install dependencies manually, jar files must be placed in the ./apps/libs folder. The jar file path must be included in apps/jetty/api/webapps/api-extra-classpath.txt and apps/jetty/connectors-classic/webapps/connectors-extra-classpath.txt.
Example

To create script objects that access and reference Java types from Javascript use the `Java.type()` function:

```javascript
var TwitterFactory = Java.type("twitter4j.TwitterFactory");
var twitter = TwitterFactorygetSingleton();
```

Debugging and Troubleshooting

To debug a JavaScript Index stage you can:

- Check the Fusion api server logs for errors.
- Use the `logger` object for print debugging (in the Fusion log file `api.log`).
- Use the Pipeline Preview tool (only available in Fusion 1.x)

The JavaScript Engine Used by Fusion

The JavaScript engine used by Fusion is the Nashorn engine from Oracle. See [The Nashorn Java API](#) for details.

Upgrading to the latest Nashorn engine

The default version of the Nashorn engine used by Fusion versions 2.4.1 and earlier is the nashorn-0.1-jdk7.jar which contains many bugs that have since been fixed in the official JDK 1.8 version. In order to use the latest version of the Nashorn engine, you must:

- Have an up-to-date version of Java 8 installed.
- Remove the nashorn-0.1-jdk7.jar from the Fusion classpaths:
  ```
  cd fusion/5.0.x
  find . -name "nashorn-0.1-jdk7.jar" -print -exec rm -i {} 
  ```

Creating and accessing Java types

The following information is taken from Oracle's JavaScript programming guide section 3, Using Java From Scripts.

To create script objects that access and reference Java types from Javascript use the `Java.type()` function:

```javascript
var ArrayList = Java.type("java.util.ArrayList");
var a = new ArrayList;
```
Stored Parameters Collection

A Stored Parameters collection contains lists of parameters, where each list is has a unique identifier called a "key". These lists of parameters can be plugged into Fusion query pipelines. The Stored Parameters facility provides a mapping layer between the keys and specific actions carried during search and is somewhat similar to database triggers or stored procedures. The use for this collection is to store Solr facet parameters for a particular document or product category, where the facet parameters are stored as a JSON object. An example of what the input JSON is shown in a section below.

A Stored Parameters collection is an auxiliary Fusion collection which is associated with a primary collection via naming conventions: for a primary collection, the name of the associated Stored Parameters collection consists of the name of the primary collection plus the suffix "_stored_parameters". This Stored Parameter collection is used by a Retrieve Stored Parameters query stage.

In the Fusion UI, Stored Parameter collections are not displayed in the list of collections in the Collections Admin panel, nor are there controls on the UI for creating or populating these collections; all collection administration is done via calls to the REST API.

Stored Parameters for Facets

• An existing facets configuration can be converted into the required format via a once-off custom script (e.g. written by Lucidworks solution engineers or by a customer's technical team).

• Note that if a caller wants particular behaviour e.g. specifying that all results should be from a given category, then it is up to them to specify that in their original request (e.g. using a filter query "fq" setting).

• A new Stored Parameter Query Stage can then be used in a query pipeline to add the required parameters for a given key (e.g. a category ID).

Stored Parameters Example

The following example shows how faceting can be carried out over a hierarchy of categories and subcategories where there are many known subcategory-specific attributes to query on. The example data is taken from the Best Buy product catalog data.

In this list of stored parameter objects, the set of fields used are:

• “id” : record ID (or "key" in the sense of a "Key-Value" store) of the category to facet on.

• “label_s” : human-readable label for this record, i.e., the category name

• “in_ids_ss” : parent categories, if an intermediate category in a category hierarchy. Intended use is in traversing a directed graph of categories.

• “out_ids_ss” : child categories, if an intermediate category in a category hierarchy. Intended use is in traversing a directed graph of categories.

• “parameters_ss” : list of arbitrary Solr parameters that can be used to construct a query that is relevant for the entity described by this record. In this example, it is the list of facet parameters to use for this specific category.
Direct Access and Navigation

A caller can query the `<collection>_stored_parameters` system collection directly.

For example, a Solr request to get the human-readable label stored in field "label_s" for a specific category ID is:

```
http://127.0.0.1:8983/solr/<collection>_stored_parameters/query?q=id:<category_id>&fl=label_s
```

Direct access allows for navigation up and down a hierarchy of categories by listing incoming and outgoing links that specify a category's location. For example, the Solr requests to find the parent(s) and children of a given category, respectively, are:

```
```

Note that this kind of "graph navigation" can become inefficient if it requires submitting a large number of queries to navigate up and down the hierarchy.
Search Applications

Ultimately, Fusion is the back end for your own search applications.

- Your application uses Fusion's REST API to interact with the Fusion system. The REST API supports all the features available in the Fusion UI. At a minimum, your application will employ the `/query-pipelines/{id}/collections/{collection}/{handler}` endpoint to query Fusion collections.

- **Recommendations** are a way to use aggregations to enhance the search experience. Based on the current search, or signals collected previously, Fusion can return results that are relevant in the end user's current context.

- Certain front-end features require some Fusion configuration:
  - Autocomplete
  - Faceting
  - Stopwords
  - Synonyms

Autocomplete

Autocomplete, also known as auto-suggest, look-ahead, or type-ahead, is a feature that displays a list of common search terms that begin with, or are contained by, the query string. For example, the query "search technology" might result in an autocomplete list with results "search technology rocks", "i love search technology", and/or "this is a search technology company", depending on how the autocomplete has been configured. Like most other search components, Fusion leverages Solr's autosuggest component. Assuming auto-suggest is configured in the Solr config, it is sufficient to enable it in Fusion in order to use the component.

How to configure the suggest component:

1. Choose a suggester implementation from the Solr Suggester documentation and define it in the solrconfig.xml file.

   In this example, a suggester called mySuggester is created in solrconfig.xml using the AnalyzingInfixLookupFactory and defining 'description' as the field to be suggested on:

   ```xml
   <searchComponent name="suggest" class="solr.SuggestComponent">
       <lst name="suggester">
           <str name="name">mySuggester</str>
           <str name="lookupImpl">AnalyzingInfixLookupFactory</str>
           <str name="dictionaryImpl">DocumentDictionaryFactory</str>
           <str name="field">description</str>
           <str name="suggestAnalyzerFieldType">string</str>
           <str name="buildOnStartup">false</str>
       </lst>
   </searchComponent>
   ```

2. Ensure that the field used for autosuggest is properly defined in the schema.

   ```xml
   <field name="description" type="text_general" stored="true" indexed="true" multivalued="false" />
   ```

3. Once modified, the custom configuration files can be uploaded to Zookeeper using Zookeeper's command line
The zkcli command `putfile` can be used to replace an existing ZK configuration file. Here `CONFIG_NAME` will likely represent the name of the collection, otherwise the name of a custom configuration that has been defined.

```
fusion/5.0.x/apps/solr-dist/server/scripts/cloud-scripts/zkcli.sh -z localhost:2181 -cmd putfile /configs/CONFIG_NAME/solrconfig.xml /Desktop/solrconfig.xml
```

4. Enabling autosuggest in Fusion requires adding the "suggest" request handler to the AllowedRequestHandlers in the "Solr Query" stage of the query pipeline. The suggester should then be built using `…/suggest?suggest.build=true`.

Example:

```
http://localhost:8764/api/query-pipelines/PIPELINE/collections/COLLECTION/suggest?suggest.build=true
```

5. Start searching!

```
http://localhost:8764/api/query-pipelines/PIPELINE/collections/COLLECTION/suggest?suggest.q=ca...
```

**Faceting**

Faceting is the name given to a set of computed counts over a search result which are returned together with the documents which match the search query. Facets are most often used to create additional navigational controls on the search results page or panel which allow users to expand and restrict their search criteria in a natural way, without having to construct complicated queries. For example, popular e-commerce facets include product category, price range, availability, and user ratings.

Fusion leverages Solr's Faceting search components.
Field faceting

In Solr the most straightforward kind of faceting is field faceting, in which Solr’s FacetComponent computes the top values for a field and returns the list of those values along with a count of the subset of documents in the search results which match that term. Field faceting works best over fields which contain a single label or set of labels from a finite, controlled lexicon such as product category. Fusion’s Facet Query Stage can be used to configure field faceting as part of the search query pipeline.

Range faceting

Range facets are used for fields which contain date or number values. Values can be grouped into ranges by specifying additional query parameters.

To configure range faceting, use the Additional Query Parameters Stage to specify Solr range faceting parameters.

Faceting concepts

Key Facet Concepts:

Term
A specific value from a field.

Limit
The maximum number of terms to be returned.

Offset
The number of top facet values to skip in the response (just like paging through search results and choosing an offset of 51 to start on page 2 when showing 50 results per page).

Sort
The order in which to list facet values: count ordering is by documents per term, descending, and index ordering is sorted on term values themselves.

Missing
The number of documents in the results set which have no value for the facet field.

Choice of facet method

(advanced) Specify Solr algorithm used to calculate facet counts. (See Facet Method Configuration for details). One of:

+ 

  • enum - small number of distinct categories
  • fc ("field cache") - many different values in the field, each document has low number of values, multi-valued field
  • fcs ("single value string fields") - good for rapidly changing indexes

Further Reading

https://lucidworks.com/blog/2014/10/03/pivot-facets-inside-and-out/

https://lucidworks.com/blog/2015/01/29/you-got-stats-in-my-facets/
Stopwords Files

Fusion collections are Solr collections that are managed by Fusion. Solr itself manages a set of resources for a collection. Stopword lists are one such resource.

Stopwords are words that are filtered out of a user-entered search request. Examples of common English stopwords are a, an, and the.

The Fusion UI provides a stopwords manager tool that is reached from the Stopwords page in the Collections menu. This tool lets you view the contents of all configured stopwords files in an editable browser.

Synonyms Files

Synonyms are words that mean the same thing, within the context where they are used. Used for search, synonym expansion allows Fusion to return results that match the meaning of the query terms, but not the words themselves. Synonyms are important for mapping query terms such as acronyms to their names, jargon to public terms, misspellings to correct spellings, old to new personal or corporate names, and otherwise bridging the gap between the user vocabulary and terms in the original text.

Fusion uses the Solr synonyms.txt and Solr collections, which are managed by Fusion. Solr itself manages a set of resources to apply synonym expansion, with configuration through the Fusion API and the Fusion UI. However, Fusion synonyms are not interchangeable with Solr synonyms files.

If you have a Fusion AI license, see also the Synonym and Similar Queries Detection job which automatically detects synonyms to use in query rewriting.

Synonym types

There are three kinds of search synonyms, depending on the requirements of the search for each specific term.

Replacement synonyms

Replacements are used to change the query, to replace it with a more standard term or terms. For example:

```
lucid => lucidworks
```

In this case, "lucid" by itself not an approved term, so there should be no instances where the company name is a partial word.

One-way expansion synonyms

Oneway expansions expand original terms with more standard terms while retaining the original term; they do not do the opposite, expand standard terms to the original non-standard terms

```
monitor => monitor, display
```

In this case, "display" is the standard term, but "monitor" is used in some older user-generated content.
Multi-way expansion synonyms

Where each term is considered equally standard, multiway synonyms expand the query so any items with any of the terms is retrieved.

login, logon, signin, signon

This example shows terms that are used interchangeably by authors. For this search engine, there is no need to distinguish among them, and considerable value in increasing recall to find all items discussing this topic, however other content stores may use them differently. Note that "logging" and "signing" have specific meanings in many contexts, so they might not be candidates for synonyms.

Example of synonym expansion:

<table>
<thead>
<tr>
<th>Results before synonyms</th>
<th>Results after synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="/home/jenkins/jenkins-slave/workspace/Fusion/Fusion-docs/Fusion-docs-publish-docker/pdfs/assets/images/common/synonyms-query-before-0-matches.png" alt="image" /></td>
<td><img src="/home/jenkins/jenkins-slave/workspace/Fusion/Fusion-docs/Fusion-docs-publish-docker/pdfs/assets/images/common/synonyms-query-after-18-matches.png" alt="image" /></td>
</tr>
</tbody>
</table>

Viewing the query using the debug=true parameter shows how it is expanded:

```
"querystring": "logon",
"parsedquery": "(+DisjunctionMaxQuery((Synonym(_text_:login _text_:logon _text_:signin _text_:signon))))/no_coord",
"parsedquery_toString": "+(Synonym(_text_:login _text_:logon _text_:signin _text_:signon))",
```

Multiword synonyms

Lucene/Solr supports multiword synonyms in version 6.6 and later, and Fusion in version 3.1 and later. There are significant technical complexities in performing graphed phrased expansion that had to be overcome.

To enable multiword synonyms in Fusion, create an Additional Parameter stage for disabling the split on whitespace tokenization process (which applies to synonyms only):

```
sow=false
```

Using EDismax, this allows the new Solr SynonymGraphFilter to create the graph representations of token streams containing overlapping synonyms of varying word counts, and expand the queries with additional terms.

Examples:

```
appstudio => app studio
signup => signup, sign up
login, log in, log on, signin, sign in, signon, sign on
```

Multiword synonyms work just like single-word synonyms, expanding the parsed query with additional query terms.
For Solr details, see: Multi-Word Synonyms: Solr Adds Query-Time Support.

**Synonyms Editor API**

The Synonyms Editor API provides complete REST access to all aspects of the synonyms.
**Solr Query Language Cheat Sheet**

This cheat sheet is a quick reference to the Solr query language. Use this syntax when querying Fusion via the Query API.

There are two ways to query a Fusion collection using the parameters below:

- Enter query parameters in the Query Workbench or the Quickstart.
- Append query parameters to the `/query/{id}` endpoint.

Wildcards and regular expressions are supported.

### Wildcards

- `?`  
  Single-character wildcard
- `*`  
  Multi-character wildcard

### Common query parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>q</code></td>
<td>The full Solr query, using Lucene query syntax.</td>
</tr>
<tr>
<td><code>fq</code></td>
<td>Filter query. A query string that limits the query results without influencing their scores.</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Sort field/direction. The field on which to sort, followed by a space and direction (desc or asc). You can specify multiple sort fields like this: <code>sort=title asc,year desc</code></td>
</tr>
<tr>
<td><code>rows</code></td>
<td>Max results per page. This set the &quot;page size&quot; for paginated search results.</td>
</tr>
<tr>
<td><code>start</code></td>
<td>Pagination offset. The number of results to skip, for pagination purposes.</td>
</tr>
<tr>
<td><code>fl</code></td>
<td>Field List. The list of fields to return in the query results.</td>
</tr>
<tr>
<td><code>wt</code></td>
<td>Response writer. Select the response format by specifying one of Solr’s response writers.</td>
</tr>
</tbody>
</table>

See also the [Solr facet parameters documentation](#).

### Query examples

Search only the `title` field in the "docs" collection for the term "solr", and format the results as JSON:

```
```
App Management

Fusion apps provide tailored search functionality to specific groups of users.

An app is a named set of linked objects, including collections, datasources, index and query pipelines, index and query profiles, parsers, and more. You can define security on a per-app basis using roles and security realms.

Generally, you will create multiple apps for different purposes. You can view all of your apps in the launcher when you log in to the Fusion UI:

To enter the Fusion workspace for any app, click the app name.

In the Fusion workspace, you can hover over the top left corner to switch to another app, create a new app, or return to the launcher:
Movie Search
TechPubs
Return to launcher
Create an app

To create an app:

1. Navigate to the launcher and click Create new app.

2. Enter an app name.

   Optionally, you can also enter a description and select the app tile color.

   If you want to choose the name of the first collection in the app, then deselect Create new app default collection. When you do this and click Create app (in the next step), Fusion Server prompts you to enter a name for the first collection. Otherwise, Fusion Server creates a default collection whose name is the same as the app name.

3. Click Create App.

   The Fusion workspace appears, and you are ready to work with your app's first collection.
Share objects between apps

Sharing objects between apps lets you reuse parts of apps and synchronize apps without duplicating objects. For example, after you have developed a query pipeline that is well-suited to your data and search application, you can reuse it in another app with a different set of datasources.

Important | When a shared object is modified in one app, the changes affect all other apps that share the object.

To share an object between apps:

1. Navigate to the app that does not yet include the object.
2. Navigate to **System > Object Explorer** or click **Find Objects**.
3. Click **In Any App**.
4. Select the object you want to include in this app.
5. Hover over the object to reveal the app icon.
6. Click the app icon.
7. Select **Add to this app**.

![Diagram of object sharing process]
Object Explorer now displays a link between the object and this app:

Now you'll see the shared object in the Fusion workspace for this app.
Export an app

To export an app, you can use either the Fusion UI or the Objects API.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>When upgrading from one Fusion version to a later one, you must use the migrator to migrate objects. The migrator automates the process of translating relevant objects to the new version. Exporting an object from one Fusion version and importing it into a different Fusion version isn’t supported.</td>
</tr>
</tbody>
</table>

Export an app with the Fusion UI

How to export an app with the Fusion UI

1. Navigate to the launcher.
2. Hover over the app you want to export and click the Configure icon:

3. In the app config window, click Export app to zip:
This downloads a zip file that you can import into other instances of Fusion Server.

**Export an app with the Objects API**

The examples below show how to export one or more apps:

Export all apps

```bash
curl -u user:pass http://localhost:8764/api/objects/export?type=app > all-apps.zip
```

Get all app IDs, then export one app by ID
> curl -u user:pass http://localhost:8764/api/apps

[{
  "id" : "movies",
  "name" : "Movies",
  "description" : "Search the movielens database.",
  "dataUri" : "/App-Tile-01-460x160.png",
  "properties" : {
    "headerImageName" : "headerImage1",
    "tileColor" : "apps-darkblue",
    "previousCollectionId" : "movies"
  }
}, {
  "id" : "tech-pubs",
  "name" : "TechPubs",
  "description" : "Search the documentation.",
  "dataUri" : "/App-Tile-02-460x160.png",
  "properties" : {
    "headerImageName" : "headerImage2",
    "tileColor" : "apps-darkblue",
    "previousCollectionId" : "tech-pubs"
  }
}]


<table>
<thead>
<tr>
<th><strong>Note</strong></th>
<th>When you export an app that includes objects that are shared with other apps, then all apps linked to that object are also exported.</th>
</tr>
</thead>
</table>

**Export two apps by ID**


<table>
<thead>
<tr>
<th><strong>Tip</strong></th>
<th>For the app object type, the <strong>deep</strong> parameter is not used. Linked objects are always included when exporting apps.</th>
</tr>
</thead>
</table>
Import an app

To import an app, you can use either the Fusion UI or the Objects API.

Import an app with the Fusion UI

How to import an app with the Fusion UI

1. Navigate to the launcher.
2. Click Import app.
3. Under Data File, click Choose File and select the zip file containing the app you want to import.
4. If your app has usernames and passwords in a separate file, select it under Variables File.

Tip

If the variables file is included inside the zip file, then you don’t need to upload it separately.
5. In some cases, you can edit parameter values to use the new values in the imported app. If this is the case, Fusion displays a dialog box that lets you edit the parameter values.

Make desired changes, and then click Import.

**Import an app with the Objects API**

Like any other Fusion object, an exported app is contained in a zip file which Fusion can consume upon import. The zip file does not need to be expanded first.

Import objects from a file and stop if there are conflicts

```bash
```

Import objects, substitute the password variables, and merge any conflicts

```bash
```

**Note**

*password_file.json* must contain plaintext passwords.

Import a zip file of Fusion objects and merge any conflicts
curl -u user:pass -H "Content-Type:multipart/form-data" -X POST -F
Delete an app

How to delete an app

1. Navigate to the launcher.

2. Click the Configure icon on the app you want to delete:

3. Click **Delete app**:

Fusion Server prompts you to confirm that you want to delete the app.

4. Click **Yes, delete**.
Template expressions are used to configure some Fusion pipeline stage and messaging services. When the value of a configuration parameter is a template expression, that expression is dynamically evaluated at runtime, by the StringTemplate library.

Fusion template expressions are delimited by angle bracket characters ‘<’ and ‘>’. The expression consists of the name of a variable in the scope of that component. Since these variables are Java objects, if object "foo" of type Foo has a field named "bar", the expression "<foo.bar>" will evaluate to the string representation of the contents of field "bar".
Index Pipeline Stage Templates

Index stages have available variables

- "ctx" - Context (com.lucidworks.apollo.pipeline.Context)
- "doc" - PipelineDocument (com.lucidworks.apollo.common.pipeline.PipelineDocument)

For example, given a PipelineDocument which has a field named "title" with value "Star Wars", the template expression "<doc.title>" evaluates to "Star Wars".
Query pipeline stage Templates

Query stages have available variables:

- “ctx” - Context (com.lucidworks.apollo.pipeline.Context)
- “reqResp” - QueryRequestAndResponse (com.lucidworks.apollo.pipeline.query.QueryRequestAndResponse)
- “request” - Query Request (com.lucidworks.apollo.pipeline.query.Request)
- “response” - Query Response (com.lucidworks.apollo.pipeline.query.Response)
Messaging Services Templates

Messages have a set of variables which correspond to the parts of a system message:

- id
- to
- from
- subject
- body
- type
- schedule

A system message is the result of evaluating the following expression:

```
{id}<to><from><subject><body><type><schedule>
```