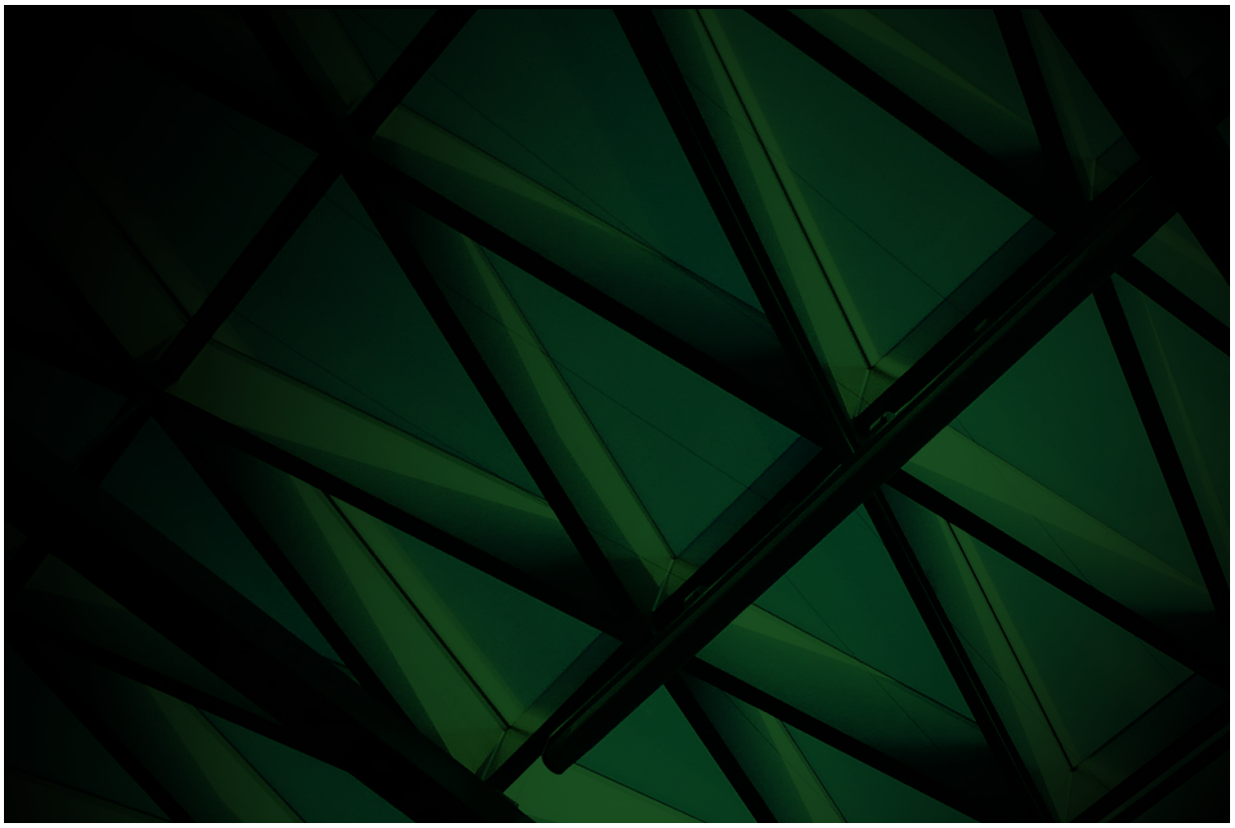


Event Streaming User's Guide

VERSION 2025.0.01.04

jade platform

Copyright©2026 Jade Software Corporation Limited. All rights reserved.

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2026 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

For details about other licensing agreements for third-party products, you must read the Jade **Readme.txt** file.

Contents

Contents ... **iii**

Before You Begin ... **vi**

Who Should Read this Guide ... vi
What's Included in this Guide ... vi
Related Documentation ... vi
Conventions ... vii

Chapter 1 Event Streaming Overview ... **8**

Concepts ... 10
Events and Event Streams ... 10
Restrictions and Limitations ... 11

Chapter 2 Quick Start Guide ... **12**

Jade Helper Scripts ... 14

Chapter 3 Configuration ... **15**

Change Capture Configuration ... 15
[EventStreaming] Section ... 15
AutoStartChangeCapture ... 15
CaptureRootDirectory ... 15
ChangeCaptureSpecificationFile ... 16
CaptureIncludeClassList ... 16
TransactionBatching ... 16
BatchSize ... 17
BatchLingerPeriod ... 17
[PersistentDb] Section ... 18
UseJournalDescriptions ... 18
EnableArchivalRecovery ... 18
JournalCloseAction ... 18
Example Jade Initialization File ... 19
Change Capture Specification File ... 19
IncludeClassList ... 19
Class Include Configuration Entry ... 20
Event Stream Producer Configuration ... 20
Configuration Structure ... 20
Modules ... 20
Configuration ... 21
KafkaSender Section ... 22
KafkaSender (for Kafka on Azure EventHubs) ... 23
Tuning Hints for KafkaSender (for Kafka on Azure EventHubs) ... 23
Event Hubs (Standard Tier) Considerations ... 24
AzureSender Section ... 24
FileSender Section ... 25
Catalog Section ... 26
SchemaCache Section ... 26
AzureSchemaRegistry Section ... 26
AzureBlobStorageProvider Section ... 27
ConfluentSchemaRegistry Section ... 28
FileReader Section ... 29
FileTopicManager Section ... 30
FileStorageProvider Section ... 30
FileStorageProvider Usage ... 30
Directory Structure ... 31
Example Configuration ... 31
Notes ... 31
SizeThresholds Section ... 32

Value Format 32
Behavior 32
Default Values 32
Notes 33
Logger Section 33
Loggers Section 34
HealthMonitor Section 34
Event Stream Producer Configuration Examples 38
Development and Testing Configuration with FileSender Example 39
File Storage Provider Configuration Example 40
Kafka Sender Configuration Example 41
Event Hub with Azure Schema Registry and File Blob Storage Configuration Example 42
Event Hub with Azure Schema Registry and Azure Blob Storage Configuration Example 43
Azure Event Hubs Configuration with Azure Schema Registry and Azure Blob Storage Example 45
Confluent Cloud with Schema Registry Configuration Example 47
OAuth2 on Confluent Cloud with Schema Registry 48
Topic Configuration 49
Configuration Structure 49
Mapping Rules 49
Class-Based Mapping 49
Event-Based Mapping 49
Special Wildcard Mapping 50
Example Configuration 50
Configuration Breakdown 50
Best Practices 51
Authentication Configuration 51
Authentication Types 51
Type-Specific Parameters 52
AzureClientSecret 52
Example Configuration 52
AzureEnvironment 52
AzureManagedIdentity 52
Event Hub and Schema Registry Requirements 52
Azure Blob Storage Requirements 53
ConnectionString 53
Example Configuration 53
SharedAccessKey 53
Example Configuration 53
ApiKey 53
Example Configuration 54
ConfluentOAuth2 54
Example Configuration 54
None 54
Example Configuration 54
Environment Variable Placeholders 55
Usage Note 55
Performing a Bulk Extract 55

Chapter 4 Installing and Running theEvent Stream Producer 58

Installing the Event Stream Producer 58
Event Stream Producer Runtime Environments 58
Scenario 1: Development and Testing with Local Single Node Kafka Cluster 59
Additional Resources 59
Scenario 2: Windows VM in Azure with Azure Event Hubs 59
Additional Resources 60
Scenario 3: Windows VM in Azure with Confluent Cloud 60
Additional Resources 60
Running Event Stream Producer Applications 60
Running the Event Stream Producer (ESP) as an Application 61
Running the ESP as a Windows Service 61

Contents v

Running the Event Schema Generator (ESG) 63
Example of Generating a Schema File for One Class 63
Example of Generating a Schema File for All Classes 63
Output Files 63
Health Monitoring Data Specification 64

Chapter 5 Event Structure and Schemas 67

Overview 67
Event Structure and Definition 67
Event Properties 68
Object Properties 69
Jade Object Properties 69
User-Defined Object Properties 69
Object Reference Properties 69
Event Schema 70
Root-Level Properties 70
Definitions Section 71
Object Definition 71
ObjectReference Definition 71
Constraints 71
JSON Schema Generation 71

Chapter 6 External Storage Providers 73

Overview 73
External Storage Provider Architecture 73
Key Features 74
Benefits 74
External Storage Providers 74
File System Provider 74
Directory Structure 75
Azure Blob Storage Provider 75

Chapter 7 Schema Registry and Caching 76

Schema Caching 76
Schema Registry 76
Azure Schema Registry 76

Appendix A Glossary 77

Before You Begin

The *Event Streaming User's Guide* is intended as the main source of information when installing and configuring the Event Stream Producer (ESP).

Who Should Read this Guide

The main audience for the *Event Streaming User's Guide* is expected to be users intending to integrate Event Streaming into their Jade Platform application or anyone interested in learning more about the capability.

What's Included in this Guide

The *Event Streaming User's Guide* has seven chapters and one appendix.

Chapter 1	Provides an introduction to event streaming
Chapter 2	Provides a quick start guide to getting started using Jade Event Streaming
Chapter 3	Provides details about configuring Change Capture, Event Stream Producer, and mapping Jade events to topics
Chapter 4	Provides details about installing and running the Event Stream Producer
Chapter 5	Provides details about the Jade event structure and schema
Chapter 6	Provides details about External Storage Providers
Chapter 7	Provides details about schema caching and schema registry
Appendix A	Glossary

Related Documentation

Other documents that are referred to in this guide, or that may be helpful, are listed in the following table, with an indication of the Jade operation or tasks to which they relate.

Title	Related to…
Developer's Reference	Developing or maintaining Jade applications
Development Environment Administration Guide	Administering the Jade Platform development environment
Jade Initialization File Reference	Maintaining Jade initialization file parameter values
Installation and Configuration Guide	Installing and configuring Jade
Object Manager Guide	Jade Object Manager administration, including security
Synchronized Database Service (SDS) Administration Guide	Administering Jade Synchronized Database Services (SDS), including Relational Population Services (RPS)

Conventions

The *Event Streaming User's Guide* uses consistent typographic conventions throughout.

Convention	Description
Arrow bullet (❯❯)	Step-by-step procedures. You can complete procedural instructions by using either the mouse or the keyboard.
Bold	Items that must be typed exactly as shown. For example, if instructed to type **foreach**, type all the bold characters exactly as they are printed.
	File, class, primitive type, method, and property names, menu commands, and dialog controls are also shown in bold type, as well as literal values stored, tested for, and sent by Jade instructions.
Italic	Parameter values or placeholders for information that must be provided; for example, if instructed to enter *class-name*, type the actual name of the class instead of the word or words shown in italic type.
	Italic type also signals a new term. An explanation accompanies the italicized type.
	Document titles and status and error messages are also shown in italic type.
Blue text	Enables you to click anywhere on the cross-reference text (the cursor symbol changes from an open hand to a hand with the index finger extended) to take you straight to that topic. For example, click on the "Event Stream Producer Configuration" cross-reference to display that topic.
Bracket symbols ([ ])	Indicate optional items.
Vertical bar (|)	Separates alternative items.
`Monospaced font`	Syntax, code examples, and error and status message text.
ALL CAPITALS	Directory names, commands, and acronyms.
Small font	Keyboard shortcut keys.

Key combinations and key sequences appear as follows.

Convention	Description
Key1+Key2	Press and hold down the first key and then press the second key. For example, "press Shift+F2" means to press and hold down the Shift key and press the F2 key. Then release both keys.
Key1,Key2	Press and release the first key, then press and release the second key. For example, "press Alt+F,X" means to hold down the Alt key, press the F key, and then release both keys before pressing and releasing the X key.

Chapter 1 Event Streaming Overview

Event streaming has gained popularity due to increasing demands for real-time interactivity and data-driven decision-making to leverage event streaming to power core business processes and deliver exceptional customer experiences by providing many benefits, including:

- Faster insights and actions

  Allows you to react to changes in environment, customer behavior, or market conditions in milliseconds rather than hours or days. This can lead to improved customer experience, operational efficiency, and competitive advantage.

- Modernization and integration

  Enables breaking down data silos and connecting heritage systems with modern applications, using a common event streaming platform that acts as a central nervous system for an organization.

- Greater agility and innovation

  You can develop and deploy new applications and features faster, by making use of an architecture that is based on event-driven communication. You can also experiment with new ideas and test them in real-time, without affecting your existing systems.

- Single source of truth

  You can store all your events in a durable, scalable, and secure event log that preserves the order and context of the events. You can use this event log as a single source of truth for your data, and replay or rewind the events for different purposes; for example, auditing, debugging, or analytics.

The Jade Event Stream Producer (ESP) delivers a reliable and performant Event Streaming capability for the Jade Platform, extending the value of the platform for Jade customers and partners. It is a crucial component of the Jade Platform's data processing strategy that enables real-time data capture and transmission, allowing for immediate analysis and action on events as they occur.

To get the Event Stream Producer (ESP) components, download the ZIP file from the Event Streaming tile on the **Extensions** page of the Jade World Developer Centre; that is, https://www.jadeplatform.com/developer-centre/extensions.

The ESP is designed to be configurable, allowing you to adjust settings based on your specific needs. This flexibility ensures optimal performance across various use cases.

The Jade ESP supports both the Apache Kafka application programming interface (API) and Azure Event Hubs, offering flexibility in deployment options. Key features include:

- Seamless integration with a Jade Platform database
- Support for incremental replication to multiple target databases

The key features are listed in the following table.

Feature	Description
Journal-based CDC	Captures state changes from database journals, starting from the latest Log Sequence Number (LSN) by default.

Feature	Description
Event serialization	Converts captured state changes into JSON-serialized events following a well-defined JSON schema for consistent data representation.
Event publishing	Events are published to one or more topics in an event stream, using Kafka or Azure Event Hubs.
Configurable operation	The ESP is enabled via configuration settings and supports customization, including the ability to set or reset the starting journal offset.
At-least-once delivery	The ESP ensures *at-least-once* message delivery semantics, retransmitting messages in case of failure.
Globally unique event identifiers	Each event has a unique identifier in the format <*database-unique-ID*>-<*object-OID-and-edition*>, enabling consumers to detect and manage duplicate events.
Flexible deployment	ESP can run on primary or secondary (SDS) databases, supporting a variety of deployment environments.
Large data handling	Support for handling large binary (blob) or string (slob) data via external storage. External data is referenced in events using URIs.
Schema registry support	The producer is designed to support the Confluent Schema Registry protocol for managing and validating schemas, ensuring compatibility with Azure Event Hubs.
Schema evolution	Events are described using JSON schemas, to enable data consistency across system versions and to enable easy validation.
Event producer	Component that publishes events. Events can be published to a file system, Apache Kafka, or Azure Event Hubs.
Native Azure Event Hubs Client	Lower-cost alternative to using an Azure Event Hubs Kafka endpoint.

The ESP processes captured state, serializes to events, and publishes events to one or several topics on an event stream.

- The ESP is enabled by configuration. (See "Event Stream Producer Configuration", in Chapter 3.)
- You can set or reset the starting offset in a journal, if required.
- Events are published in JSON format described using a JSON schema (https://json-schema.org).
- The ESP can run on a primary or SDS secondary database. The capture process can run on a native SDS secondary database.
- Blob and slob properties are included in parent object create, update, or delete events.
- The **JournalCloseAction** parameter with a value of **Move** in the [PersistentDb] section of the Jade initialization file is supported; that is, journal capture looks for journals in both the current and the archive journal directories.
- At-least-once message delivery semantics means that in case of failures that lead to message loss or taking too long to recover from a failure, messages are retransmitted to assure they are delivered at least once.
- Every event has a globally unique identifier that enables consumers to detect duplicate events. The format is <*database-unique-ID*>-<*object-OID-and-edition*>; for example:

```
"eventId" : "2b317b93-b2bf-ee11-90bd-005056b64821.2429.1.2"
```

- The ESP implementation supports representing large data; for example, blobs and slobs in external storage with a URI that links to data in the main event.
- The ESP supports Confluent Schema Registry and Azure Schema Registry for Event Hubs.
- The ESP handles errors for asynchronous requests.

Concepts

The Event Stream Producer (ESP) is a key component of the Jade Platform, enabling real-time data capture and transmission for downstream processing. It uses a journal-based Change Data Capture (CDC) mechanism to record state changes from the database and publish events to one or multiple event streams; for example, to Kafka or Azure Event Hubs.

Events and Event Streams

An *event* is a record of a state change in the system.

An *event stream* is a sequence of events that flow from publishers to subscribers.

In a payment processing use case, an event stream could be the continuous flow of information about completed financial transactions.

In an Enterprise Resource Planning (ERP) use case, an event stream could be the continuous flow of information about product price changes.

The technology used to store and manage the stream is an event streaming platform. The two main platforms Jade Event Streaming is targeting are Kafka and Azure Event Hubs.

Jade Platform Event Streaming provides:

- Change Data Capture (CDC), which captures state changes from database journals in a temporary local data store for downstream processing, converts them to events, and writes them to a stream. Event capture starts from the tail of the current database journal, by default, but a starting Log Sequence Number (LSN) can also be specified.
  - Captures database state changes into a local data store.
  - Runs on a non-SDS database, SDS primary database, or native SDS secondary database.
  - Uses an Event Stream Producer (ESP) to serialize events into JSON format.
  - Supports JSON schema for event descriptions.
- ESP features
  - Configuration for class-level inclusion list.
  - Setting or resetting the starting journal and offset.
  - Can run on a primary or secondary database but not an RPS secondary database.
- Message delivery
  - Guarantees at-least-once delivery semantics for messages.

- Kafka integration
  - Tested with Apache Kafka and Azure Event Hubs with Kafka interface.
- Native Azure Event Hubs support (a lower-cost alternative to using an Azure Event Hubs Kafka endpoint)

Restrictions and Limitations

The Event Stream Producer (ESP) restrictions and limitations are as follows.

- Restrictions
  - Due to the inability to decrypt journalled data changes, the ESP cannot run on an encrypted database.
  - The ESP cannot run on an RPS secondary database. This is a permanent restriction.
  - Dynamic properties are not captured.
  - The Change Capture process can only be run against the primary or a native secondary database, not against both. This is a permanent restriction.
- Limitations
  - Delta database updates are not captured

    The ESP does not capture updates made to delta databases. This is a temporary limitation.
  - Primitive array updates are not captured

    Updates made to primitive arrays (for example, arrays of integers, strings, or other primitive types) are not captured by the ESP.
  - **Integer64** range in JSON

    The Jade **Integer64** primitive type maps to the JSON type integer, which has a portable range limit of **-2^53** through **2^53**. Although this is not a constraint of the JSON specification or some parsers, intermediate components based on JavaScript may impose this as an effective boundary.

    Because of this, when using this type with very large positive or negative integer values (for example, values outside the range **-2^53** through **2^53**) or for alternative purposes like hash values or bit-wise flags, it is important to consider this limitation. In such cases, encoding the property as a string may be more suitable.
  - Collection updates are not captured directly

    Updates made to collections (for example, object arrays, dictionaries, and so on) are not captured by the ESP directly. If the collection is used to implement a one-to-many inverse relationship then the information it contains will be published as part of the participating objects of the inverse if they are configured to be published as events.

Chapter 2 Quick Start Guide

This chapter provides a quick guide to getting started using Jade Event Streaming, as follows.

1. Obtain the Event Stream Producer (ESP) components from the Event Streaming tile on the **Extensions** page of the Jade World Developer Centre; that is, https://www.jadeplatform.com/developer-centre/extensions.

   The .**zip** file contains the following executable files and supporting libraries.

   - **JadeEventProducer.exe** (Event Stream Producer)
   - **JadeEventSchemaGenerator.exe** (Jade Event Schema Generator utility)

2. Install the Event Stream Producer.

   a. Extract the .**zip** file contents to a suitable directory, but *not* to the main **Jade\bin** directory.

   b. Ensure the ESP has access to change capture output files created by the source Jade system.

   c. For first-time use and initial testing, we recommend installing the ESP on the same machine as the source Jade system.

3. Run ESP applications.

   a. Running the Event Stream Producer (ESP)

   The syntax for running the ESP is as follows.

   ```
   JadeEventProducer <config-file-name> [Producer Name]
   ```

   The **JadeEventProducer** parameters are listed in the following table.

Parameter	Description
<config-file-name>	Path to the event stream producer configuration file.
Producer name	Optional name for this producer. This is represented in the log file and is useful when distinguishing between messages from producers running on different databases. The default producer name is **DEFAULT**. If you try to run more than one producer reading from the same change capture directory with the same name, this will be detected, logged, and the producer shut down.

   The following is an example of running the ESP.

   ```
   JadeEventProducer.exe config\producer.config.json PRODUCER1
   ```

   Refer to "Event Stream Producer Configuration" in Chapter 3 for details about setting up the **Producer.config.json** file.

   b. Running the Event Schema Generator (ESG)

   The **JadeEventSchemaGenerator** is a useful utility for previewing the Event Schema file that corresponds to a specific **Event** class. The **JadeEventProducer** itself automatically generates and registers these Event Schema files on demand with the "**SchemaRegistry**" configuration section if configured to do so. For details, see "Modules", in Chapter 3.

The syntax of the **JadeEventSchemaGenerator** ESG command is as follows.

```
JadeEventSchemaGenerator <input-file-name> <output-root-folder>
[<schema-name>] [<class-name>]
```

The **JadeEventSchemaGenerator** command parameters are listed in the following table.

Parameter	Description
<input-file-name>	Path to the Jade Audit Access description file (input)
<output-root-folder>	Directory where generated JSON schema files will be saved
<schema-name>	Optional name of the schema containing the class to be processed
<class-name>	Optional name of the class for which to generate a specification

Examples

The following example generates a single JSON schema file for a single class.

```
JadeEventSchemaGenerator description20240725135130.txt ./event-schemas
MySchema MyClass
```

This example generates a single JSON schema file for the specified class and saves it in **./event-schemas/MySchema/**. The generated file is named **MyClass_v0001.schema.json** (for version 0001 of the class).

The following example generates a JSON schema file for each user class and saves them in the **./event-schemas/** folder structure.

```
JadeEventSchemaGenerator description20240725135130.txt ./event-schemas
```

This example generates a JSON schema file for each user class and saves them in the **./event-schemas/** folder structure.

Note All collection classes and system classes are ignored.

Output File Structure

The output file structure is as follows.

```
<output-root-folder>
└── <schema-name>
    └── <class-name>_v<class-version>.schema.json
```

4. Next steps.

   a. Review the configuration, by referring to Chapter 3, Configuration, for details about setting up the **Producer.config.json** file.

   b. For details about advanced usage and troubleshooting, see subsequent chapters of this guide.

   c. For details about performing a bulk extract, see "Performing a Bulk Extract", in Chapter 3.

Jade Helper Scripts

Generate a journal description file, as follows.

```
vars
    jaa : JadeAuditAccess;
begin
    create jaa transient;
    beginTransaction;
    write jaa.descriptionPath;
    jaa.generateDescription();
    commitTransaction;
epilog
    delete jaa;
end;
```

Stop journal capture, as follows.

```
stopCapture();
begin
    system.stopJournalChangeCapture();
end;
```

Set (or reset) the change capture journal and journal offset, as follows.

```
setCaptureState();
vars
   journalNumber : Integer64;
   offset        : Integer64;
begin
    journalNumber := 56382;
    offset := 1;
    system.setJournalChangeCaptureState(journalNumber, offset);
end;
```

Start journal capture (when stopped), as follows.

```
startCapture();
begin
    system.startJournalChangeCapture();
end;
```

Close the current journal, as follows.

```
closeJournal();
vars
    dba : JadeDatabaseAdmin;
begin
    create dba transient;
    dba.closeCurrentJournal();
    write dba.getCurrentJournalName();
epilog
    delete dba;
end;
```

Chapter 3 Configuration

This chapter covers the following topics.

- Change Capture Configuration
- Event Stream Producer Configuration
- Topic Configuration
- Authentication Configuration
- Performing a Bulk Extract

Change Capture Configuration

This topic contains the settings for configuring change capture in the Jade initialization file (**jade.ini**, by default). The settings are organized by section, with details about each parameter in the relevant section.

For details about any other Jade initialization sections or parameters not documented in this topic, see the *Jade Initialization File Reference* in the Jade Platform product information library, available at:

https://secure.jadeworld.com/developer-centre/Jade2025/OnlineDocumentation/

[EventStreaming] Section

This section contains settings specific to event streaming and change capture functionality.

AutoStartChangeCapture

```
AutoStartChangeCapture=true|false
```

Value type: Boolean

Default: false

Values: The default value of **false** does *not* automatically start the Change Capture server application. Set this parameter to **true** if you want the Change Capture server application to start automatically.

Purpose: Controls the automatic start of the Journal Change Capture server application.

Behavior: When set to **true**, the Change Capture server application starts automatically when the database server node is started. This setting applies to multiuser mode only.

CaptureRootDirectory

```
CaptureRootDirectory=<root-directory-path>
```

Value type: String

Value: A valid directory path; for example, **d:\ChangeCaptureOutput**.

Purpose: Specifies the root directory for event capture files.

Behavior: The specified directory is shared with the Event Stream Producer (ESP).

A **ChangeCapture** subdirectory is automatically created under this root directory. Under the **ChangeCapture** subdirectory, the following subdirectories are created.

- Staging
- Processing
- Processed

ChangeCaptureSpecificationFile

```
ChangeCaptureSpecificationFile=C:\jade\JadeChangeCapture.config.json
```

Purpose: Specifies the location of the Change Capture specification file.

Format: A valid file path; for example, **C:\jade\JadeChangeCapture.config.json**.

Behavior: Specifies where the Change Capture specification file is located. The Change Capture specification file enables you to configure which classes are included in the change capture. For details, see "Change Capture Specification File", later in this chapter.

Note If you specify a value for this parameter, it will override the **CaptureIncludeClassList** parameter.

CaptureIncludeClassList

```
CaptureIncludeClassList=<list-of-classes-to-include>
```

Value type: String

Purpose: Specifies the list of classes to include for capture.

Format: Comma-separated list of ***<schema-name>*::*<class-name>*** pairs; for example,

```
CaptureClassList=BankModelSchema::Account,BankModelSchema::Customer,
BankModelSchema::District,BankModelSchema::Transaction
```

Behavior: Only the classes specified in this list are included in the change capture process.

Note If the value of the **ChangeCaptureSpecificationFile** parameter is **true**, this parameter will be overwritten by the configuration in that file. If the **ChangeCaptureSpecificationFile** option is unspecified, and this option is unspecified or does not contain any valid class names, exception 8651 (*Jade Change Capture must be configured to capture at least one class*) is raised.

TransactionBatching

```
TransactionBatching=true|false
```

Value type: Boolean

Default: false

Value: The default value of **false** does *not* start the Change Capture server application in transaction batching mode. In non-transaction batching mode, each transaction is written to its own file (legacy behavior)

Set this parameter to **true** if you want the Change Capture server application to start up in transaction batching mode. In this mode, batches of committed transactions are written to a multi-transaction capture file with a header, in ascending order, asynchronously when close to the specified **BatchSize** or if the **BatchLingerPeriod** period has been reached.

Purpose: Controls how events are written to Change Capture files.

Behavior: When this optional parameter is set to **true**, the Change Capture server application starts up in transaction batching mode. When running in this mode, multiple transactions are written to Change Capture files.

BatchSize

```
BatchSize=<file-size>KB|MB|GB
```

Value type: Integer (prefix multiplier)

Default: 500KB

Value: The target Change Capture batch file size specified in kilobytes(KB), megabytes(MB), or gigabytes (GB).

Purpose: Controls the target file size of Change Capture files when running in transaction batching mode. This means the number of transactions in each batch varies, based on the size of the transactions.

Behavior: When event batching is enabled, this parameter specifies the target size of the Change Capture files. The Change Capture server application batches up transactions into a Change Capture file until the specified file size is met or the specified **BatchLingerPeriod** elapses.

If you change the value of this optional parameter, the Change Capture application server must be restarted for the changes to take effect.

If transaction batching is enabled and no value is specified, this parameter defaults to **500KB**, which is the recommended target file size.

Note If targeting Azure Event Hubs, this value is recommended to be less than 1MB, as that is the maximum message size when sending to an EventHub.

BatchLingerPeriod

```
BatchLingerPeriod=200
```

Value type: Integer

Default: 200

Value: The length of time specified in milliseconds (ms). (A unit is not required for this parameter.)

Purpose: Controls how long to keep the file open before moving on to a new file.

Behavior: When event batching is enabled, this parameter specifies the target linger period. The Change Capture server application batches up transactions into a Change Capture file until the specified period elapses or the specified **BatchSize** is met.

If you change the value of this optional parameter, ChangeCapture application server must be restarted for the changes to take effect.

If transaction batching is enabled and no value is specified, this parameter defaults to **200** milliseconds.

[PersistentDb] Section

This section contains parameters related to the persistent database and journal management required for event capture.

UseJournalDescriptions

```
UseJournalDescriptions=true|false
```

Value: Boolean

Default: Not specified

Example:

```
UseJournalDescriptions=true
```

Behavior: To support generation of event schemas, the **UseJournalDescriptions** parameter must be set to **true**. When enabled, a new journal description file is generated each time you migrate a schema (or you execute a reorganization).

Event schemas are generated only when the **UseJournalDescriptions** parameter exists in the Jade initialization file and it is set to **true**.

EnableArchivalRecovery

```
EnableArchivalRecovery=true|false
```

Value: Boolean

Default: Not specified

Example:

```
EnableArchivalRecovery=true
```

Behavior: This parameter enables or disables the archival recovery option.

It is essential that you enable this parameter (that is, set it to **true**) so that database journals required for processing by the Change Capture server application are retained after they have been released.

Note This must be set to **true** (that is, enabled) for a non-SDS primary database.

JournalCloseAction

```
JournalCloseAction=None
```

Value: Boolean

Default: Move

Example:

```
JournalCloseAction=Move
```

Behavior: This parameter specifies what to do for transaction abort processing or restart recovery processing when a journal file that is no longer required is closed and released.

When archival recovery is enabled, the close actions that you can specify are listed in the following table.

Journal Close Action	Processing
None	No action taken.
Move	Uses the operating system to move the file to the archive directory.

The parameter is read when the database server node is initialized; for example, when you restart the database server.

Example Jade Initialization File

```
[PersistentDb]
UseJournalDescriptions=true
EnableArchivalRecovery=true
JournalCloseAction=None

[EventStreaming]
AutoStartChangeCapture=true
CaptureIncludeClassList=BankModelSchema::Account,BankModelSchema::Customer,
BankModelSchema::District,BankModelSchema::Transaction
CaptureRootDirectory=d:\ChangeCaptureOutput
```

Change Capture Specification File

The Change Capture specification file contains user-defined JSON data that specifies which classes should be included in the change capture.

The following is an example of a Change Capture specification file for two classes.

```
{
  "IncludeClassList": [
    {
      "name" : "BankModelSchema::Bank"
    },
  {
      "name" : "BankModelSchema:Customer",
      "recursiveSubclasses" : true
    }
  ]
}
```

The configuration consists of a top-level object which contains the configuration elements. The currently available configuration elements are described in the following sections.

IncludeClassList

The **IncludeClassList** section contains the Class Include Configuration Entries that are the classes to include in the change capture.

There is no limit on the number of Class Include Configuration Entries you can specify.

If the **IncludeClassList** section is unspecified or it does not contain any valid Class Include Configuration Entries, exception 8651 (*Jade Change Capture must be configured to capture at least one class*) is raised.

Class Include Configuration Entry

The Class Include Configuration Entry values are listed in the following table.

Value	Required?	Description
name	Yes	Specifies the name of the class to include. The **name** value format is ***<schema-name>::<class-name>***. This value is mandatory for a Class Include Configuration Entry. If it is not specified or the name does not match an existing class, the entry will be invalid.
recursiveSubclasses	No	Specifies whether subclasses of this class should also be included recursively; that is, all subclasses of this class, the subclasses of those classes, and so on. This does not include subclasses existing within another schema or subschema copies of this class. This value is optional and if unspecified, defaults to **false**.

Event Stream Producer Configuration

This topic describes the configuration elements for the Event Stream Producer (ESP) configuration file and how to obtain the required values.

Configuration Structure

The configuration structure is as follows.

```
{
  "Modules": { ... },
  "Configurations": { ... }
}
```

For details, see "Modules" and "Configuration", in the following sections. See also "Authentication Configuration" and "Event Stream Producer Configuration Examples", elsewhere in this chapter.

Modules

The "**Modules**" section defines the types of components to use, as follows.

```
"Modules" : {
  "Reader" : "File",
  "Sender" : "<SenderType>",
  "Serializer" : "Json",
  "StorageProvider" : "<StorageProviderType>",
  "TopicManager" : "File",
  "SchemaGenerator" : "Json",
  "SchemaRegistry" : "<SchemaRegistryType>"
}
```

The **Modules** section components are listed in the following table.

Component	Type of...
"Reader"	Reader to use, which must be **"File"** in this release.
"Sender"	Sender to use, which can be one of: **"Kafka"**, **"Azure"**, or **"File"**. These are referred to as **SenderType**.
"Serializer"	Type of serializer to use. This must be **"Json"**.
"StorageProvider"	Type of external storage provider to use for storing over-threshold size blob and slob data, which can be one of: **"File"** or **"AzureBlobStorage"**.
"TopicManager"	Topic manager to use, which must be **"File"** in this release.
"SchemaGenerator"	Type of schema generator to use. This must be **"Json"**.
"SchemaRegistry"	Type of "**SchemaRegistry**" to use for looking up and storing Event Schemas. This can be one of: **"None"** or **"Azure"**, or **"Confluent"**.

The **SenderType** essentially selects the event streaming platform (or file sender for development and testing) and the appropriate client for the selected type. The selected **SenderType** determines which **<SenderType>Sender** section in the **Configurations** section will be used.

Note All required modules must be specified in the **Modules** section. The ESP checks for the presence of all seven modules; that is, **Reader**, **Sender**, **TopicManager**, **Serializer**, **StorageProvider**, **SchemaGenerator**, and **SchemaRegistry**.

Configuration

Each subsection in "**Configurations**" corresponds to a specific module type and contains its configuration parameters.

The Producer configuration files use JSON format and allow single- and multiple-line comments, as follows.

- Single-line comments begin with double stroke character string (*//*).

  This type of comment continues only until the end of the line on which it appears.

- Multiple-line comments are enclosed with stroke asterisk (*/**) and asterisk stroke (***/*) character strings.

  This type of comment can span multiple lines.

Only JSON configurations files support comments not JSON Schema or Event Data files as these are primarily data exchange formats. If you use comments in your JSON configuration files, we recommend that you save these files as the .**jsonc** file type; for example, **Producer.config.jsonc**.

The subsections in the "**Configurations**" section are as follows.

```
"Configurations" : {
   "AzureSender" : { ... },
   "FileSender" : { ... },
   "Logger" : { ... },
   "Loggers" : [],
   "FileReader" : { ... },
   "FileTopicManager" : { ... },
   "KafkaSender" : { ... },
   "FileStorageProvider" : { ... },
   "SizeThresholds" : { ... },
```

```
    "Catalog" : { ... },
    "SchemaCache" : { ... },
    "AzureSchemaRegistry" : { ... },
    "AzureBlobStorageProvider" : { ... },
    "ConfluentSchemaRegistry" : { ... },
    "HealthMonitor" : { ... },
}
```

For details about configuration sections, see the following subsections.

- KafkaSender Section
- KafkaSender for Azure Section
- AzureSender Section
- FileSender Section
- Catalog Section
- SchemaCache Section
- AzureSchemaRegistry Section
- AzureBlobStorageProvider Section
- ConfluentSchemaRegistry Section
- FileReader Section
- FileTopicManager Section
- FileStorageProvider Section
- SizeThresholds Section
- Logger Section
- Loggers Section
- HealthMonitor Section

KafkaSender Section

The "**KafkaSender**" section provides configuration settings for any **Sender** producing through a Kafka API (for example, a local Apache Kafka or Azure Event Hubs with a Kafka Surface) that connects to a Kafka broker.

```
"KafkaSender" : {
   "Properties" : {
      "bootstrap.servers" : "<kafka-broker-address>",
      "security.protocol" : "<security-protocol>"
   }
}
```

The "**KafkaSender**" "**Properties**" parameters are listed in the following table.

Parameter	Required?	Description
bootstrap.servers	Yes	Address of the Kafka broker or brokers
security.protocol	No	Security protocol for Kafka (for example, **"plaintext"**, **"SSL"**, **"SASL_PLAINTEXT"**, or **"SASL_SSL"**)

See https://docs.confluent.io/platform/current/clients/librdkafka/html/md_CONFIGURATION.html for the full list of Kafka properties that can be specified. Of these, the only required setting is **bootstrap.servers**.

KafkaSender (for Kafka on Azure EventHubs)

The "**KafkaSender**" for Azure section provides configuration settings for a Kafka client using the Azure Event Hubs Kafka Surface endpoint for sending Events to Azure Event Hubs. It requires an Event Hubs namespace with Standard tier or higher.

```
"KafkaSender" : {
  "Properties" : {
    "bootstrap.servers" : "<namespace-name>.servicebus.windows.net:9093",
    "sasl.mechanisms" : "PLAIN",
    "sasl.password" : "Endpoint=sb://<namespace-name>.servicebus.windows.net/;
     SharedAccessKeyName=<policy-name>;
     SharedAccessKey=<shared-access-key>",
    "sasl.username" : "$ConnectionString",
    "security.protocol" : "SASL_SSL"
  }
}
```

The "**KafkaSender**" for Azure "**Properties**" parameters are listed in the following table.

Parameter	Description
bootstrap.servers	The Event Hubs namespace name followed by **.servicebus.windows.net:9093**
sasl.mechanisms	Always **"PLAIN"** for Event Hubs
sasl.password	The full connection string for the Event Hubs namespace
sasl.username	Always **"$ConnectionString"** for Event Hubs
security.protocol	Always **"SASL_SSL"** for Event Hubs

Tuning Hints for KafkaSender (for Kafka on Azure EventHubs)

Event Hubs (Standard tier) enforces limits on message size, data rate, and message rates. Surpassing these limits could result in failed event delivery. To mitigate this, tune the **KafkaSender** settings as follows.

```
{
   "KafkaSender" : {
      "Properties" : {
         "bootstrap.servers" : "example-eventhubs-namespace.
          servicebus.windows.net:9093",
         "connections.max.idle.ms" : "180000",
         "enable.idempotence" : "true",
         "max.request.size" : "1000000",
         "metadata.max.age.ms" : "180000",
         "request.timeout.ms" : "30000",
```

```
            "retries" : "2147483647",
            "retry.backoff.max.ms" : "300000",
            "retry.backoff.ms" : "1000",
            "sasl.mechanisms" : "PLAIN",
            "sasl.password" : "Endpoint=sb://example-eventhubs-namespace.
             servicebus.windows.net/;SharedAccessKeyName=KeyName;
             SharedAccessKey=abcdef2321312312=",
            "sasl.username" : "$ConnectionString",
            "security.protocol" : "SASL_SSL",
            "socket.keepalive.enable" : "true"
        }
    }
}
```

Key settings are explained in the following table.

Setting	Explanation
retries	Set to a large number (for example, **2147483647**) to ensure continuous retries if Event Hubs throttles requests.
retry.backoff.ms and retry.backoff.max.ms	Introduces a delay between retries to prevent overwhelming the server. Initial retry delay starts at 1000 ms, increasing to a maximum of 300000 ms.
enable.idempotence	Ensures events are delivered in order, even after retries, which is crucial when Event Hubs throttles requests.
max.request.size	Configures the maximum size of a Kafka request. Set to avoid exceeding the limits imposed by Event Hubs.
request.timeout.ms	Timeout value to allow for longer waits during retries, preventing premature failures.

Event Hubs (Standard Tier) Considerations

Without the appropriate settings, Event Hubs (Standard tier) may reject events that exceed its message or data rate limits. In such cases, manual intervention is needed to move unsent event data files from the **Undeliverable** folder to the **ChangeCapture** folder for retry.

AzureSender Section

The "**AzureSender**" section provides configuration settings for a native Azure EventHubs client. The Azure EventHubs namespace must have been set up previously with individual Event Hubs defined for each topic to be used.

```
"AzureSender" : {
  "EventHubsNamespace" : "<event-hubs-connection-namespace>",
  "NumberOfSenders": 3,
  "Authentication" : {
    "Type" : "AzureClientSecret",
    "TenantId" : "<azure-ad-tenant-id>",
    "ClientId" : "<azure-ad-client-id>",
    "ClientSecret" : "<azure-ad-client-secret>" }
```

Note You can configure a check for stalled senders in the **HealthMonitor** section.

The "**AzureSender**" parameters are listed in the following table.

Parameter	Description
EventHubsNamespace	The fully-qualified namespace for the EventHubs Namespace. (Not required if the **Authentication** parameter is **"ConnectionString"**.)
NumberOfSenders	The number of events able to be sent to an EventHub simultaneously. The optimum number of senders is determined by the partition count of an EventHub and also the distribution of EventHubs. If all events are being sent to the same EventHub, setting the number of senders to more than the partition count may not improve performance. The **NumberOfSenders** parameter value must be in the range 1 through 32 (inclusive). The default value is **1**.
Authentication	The **AzureSender** section **Authentication** parameter **Type** can be one of the following. ■ **ConnectionString** ■ **AzureLogin** ■ **AzureClientSecret** ■ **AzureEnvironment** ■ **AzureManagedIdentity** ■ **SharedAccessKey** For details, see "Type-Specific Parameters", later in this chapter.

An example of the **"ConnectionString"** parameter is as follows.

```
"AzureSender" : {
  "EventHubsNamespace" : "<event-hubs-connection-namespace>",
  "NumberOfSenders": 3,
  "Authentication" : {
    "Type" : "ConnectionString",
    "ConnectionString" : "Endpoint=sb://myeventhubs.servicebus.windows.net/;
                          SharedAccessKeyName=RootManageSharedAccessKey;
                          SharedAccessKey=abc123..." }
```

FileSender Section

The "**FileSender**" section provides configuration settings for a sender that writes events to a local file system. The sender creates subdirectories under the specified output directory, one for each configured topic.

This option is intended for development and testing scenarios, avoiding the need for access to Kafka or Event Hubs.

```
"FileSender" : {
   "OutputDirectory" : "D:\\FileCaptureOutput"
}
```

The **OutputDirectory** parameter specifies the directory path where the **FileSender** writes output files.

Note File paths in this type of configuration require double back slash characters (\\), which to be valid JSON must be escaped with a back slash (\); otherwise it fails to parse.

Catalog Section

The "**Catalog**" section provides configuration for the **Catalog** module.

This configuration specifies the directory that Jade Audit Access description files are located; for example, **description{timestamp}.txt** files. The **Catalog** module uses this information to provide the ESP with type and property information as well as in the generation of the Event Schema files.

```
"Catalog" : {
   "CatalogDirectory" : "<producer-catalog-directory>"
}
```

SchemaCache Section

The "**SchemaCache**" section provides configuration for the **SchemaCache** module.

This configuration specifies the root directory under which cached copies of the Event Schema files are to be stored locally. These files are copies of those already registered in the **SchemaRegistry**. It can also store persistently other information associated with the **SchemaCache**.

```
"SchemaCache" : {
   "CacheRootDirectory" : "<producer-cache-root-directory>"
}
```

The output files are organized under the following folder structure.

```
<producer-cache-root-directory>
   <schema-name>
   <schema-name>
```

Each JSON Schema file is stored in the subfolder for its respective Jade schema and has the following file name.

```
<class-name>_v<class-version>.schema.json
```

AzureSchemaRegistry Section

The "**AzureSchemaRegistry**" section provides configuration for the Azure Schema Registry used for registering Event Schemas that form the data contract.

```
"AzureSchemaRegistry" : {
   "Endpoint" : "https://<namespace>.servicebus.windows.net/",
   "SchemaGroup" : "<schema-group>",
   "SchemaFormat" : "Json",
   "Authentication" : {
      "Type" : "AzureLogin"
   },
   "RetryOptions" : {
      "MaxRetries" : 3,
      "RetryDelay" : 800,
      "MaxRetryDelay" : 60000,
      "StatusCodes" : [408, 500, 502, 503, 504],
   },
   "EventhubNamespace" : "<eventhubnamespace>"
)
```

Note Only one of the "**ConfluentSchemaRegistry**" or "**AzureSchemaRegistry**" sections is expected, based on the ***SchemaRegistryType*** specified in the "**Modules**" section.

The "**AzureSchemaRegistry**" parameters are listed in the following table.

Parameter	Required?	Description
Endpoint	Yes	URL endpoint for the Azure SchemaRegistry service, in the form **https://<*namespace*>.servicebus.windows.net/**.
SchemaGroup	Yes	Name of the schema group set up under the specified namespace.
SchemaFormat	Yes	Format that the schema files describe. Currently only **"Json"** is supported.
Authentication	Yes	The **AzureSchemaRegistry** section **Authentication** parameter **Type** can be one of the following. ■ **AzureLogin** ■ **AzureClientSecret** ■ **AzureEnvironment** ■ **AzureManagedIdentity** For details, see "Type-Specific Parameters", later in this chapter.
RetryOptions	No	Can be used to configure the retry behavior of the transport mechanism used to communicate with the Schema Registry. It allows fine-tuning of the retry mechanism to make the availability of the service more resilient. The retry options are: ■ "**MaxRetries**" is the maximum number of retry attempts before reporting an error. The default value is **3**. ■ "**RetryDelay**" is the minimum delay in milliseconds between retry attempts. The default value is **800** milliseconds. ■ "**MaxRetryDelay**" is the maximum delay in milliseconds between retry attempts. The default value is **60** seconds. ■ "**StatusCodes**" is the HTTP status codes that indicate when an operation should be retried. The default values are HTTP status codes RequestTimeout(**408**), InternalServerError(**500**), BadGateway(**502**), ServiceUnavailable(**503**), GatewayTimeout (**504**).
EventhubNamespace	No	Reserved for future use.

AzureBlobStorageProvider Section

The "**AzureBlobStorageProvider**" section provides configuration for the Azure Blob Storage provider, used for storing over-threshold size blob and slob data in Azure Blob Storage.

```
"AzureBlobStorageProvider" : {
   "Endpoint" : "https://<storage-account-name>.blob.core.windows.net",
   "ContainerName" : "<container-name>",
   "StagingDirectory" : "<staging-root-directory>",
   "BlobPrefix" : "<optional-prefix>",
   "Authentication" : {
      "Type : "ConnectionString",
      "ConnectionString" : "DefaultEndpointsProtocol=https;
                            AccountName=<storage-account-name>;
                            AccountKey=<account-key>;
```

```
                                    EndpointSuffix=core.windows.net"}
        }
```

The parameters are listed in the following table.

Parameter	Required?	Description
Endpoint	Yes	URL endpoint of the Azure Blob Storage service for your account, which is typically in the format **https://<*storage-account-name*>.blob.core.windows.net**. (Not required if the **Authentication** type is **"ConnectionString"**.)
ContainerName	Yes	Name of the blob container within your storage account.
StagingDirectory	Yes	Staging directory where blob and slob files are stored before they are uploaded to Azure Blob Storage. A directory hierarchy similar to that used by the **FileStorageProvider** is created under the specified **StagingDirectory**. Staged capture files are removed after they have been uploaded to Azure Blob Storage. **Notes** This unique directory name cannot be shared if multiple **JadeEventProducer.exe** applications are run. As the temporary directories created to house the blob files are not cleaned up by the application, you need to maintain them yourself at this stage.
BlobPrefix	No	Optional prefix added to all blob directory names; for example, **"events"**.
Authentication	Yes	The **BlobStorageProvider** section **Authentication** parameter **Type** can be one of the following. ■ **AzureLogin** ■ **AzureClientSecret** ■ **AzureEnvironment** ■ **AzureManagedIdentity** ■ **ConnectionString** For details, see "Type-Specific Parameters", later in this chapter.

For Azure-related configurations, use the Azure Portal or Azure Command Line Interface (CLI).

ConfluentSchemaRegistry Section

The "**ConfluentSchemaRegistry**" section provides configuration for the centralized repository for managing and validating schemas used for topic message data, as well as for the serialization and deserialization of data over the network.

```
"ConfluentSchemaRegistry" : {
   "Endpoint" : "<confluent-schema-registry-endpoint>",
   "SchemaFormat" : "Json",
   "Authentication" : {
      "Type" : "ApiKey",
      "ApiKeyName" : "1234567890",
      "ApiKeyValue" : "ABCDEFEGGHIJKLMNOPQRSTUVWXYZ1234567890+" },
```

```
      "RetryOptions" : { "MaxRetries" : 3 }
   }
```

Note Only one of the "**ConfluentSchemaRegistry**" or "**AzureSchemaRegistry**" sections is expected, based on the ***SchemaRegistryType*** specified in the "**Modules**" section.

The "**ConfluentSchemaRegistry**" section parameters are listed in the following table.

Parameter	Required?	Description
Endpoint	Yes	URL endpoint of your Confluent Schema Registry, which is typically in the format **https://<*your-namespace*>.confluent.cloud**.
SchemaFormat	Yes	Format that the schema files describe. Currently only **"Json"** is supported.
Authentication	Yes	The **ConfluentSchemaRegistry** section **Authentication** parameter **Type** can be one of the following. ■ **ApiKey** **Note** The **ApiKey** is different from the one used to connect to the Confluent Kafka Sender. ■ **ConfluentOAuth2** For details, see "Type-Specific Parameters", later in this chapter.
RetryOptions	No	Can be used to configure the retry behavior of the transport mechanism used to communicate with the Confluent Schema Registry. It allows fine-tuning of the retry mechanism to make the availability of the service more resilient. The retry options are: ■ "**MaxRetries**" is the maximum number of retry attempts before reporting an error. The default value is **3**. ■ "**RetryDelay**" is the minimum delay in milliseconds between retry attempts. The default value is **800** milliseconds. ■ "**MaxRetryDelay**" is the maximum delay in milliseconds between retry attempts. The default value is **60** seconds. ■ "**StatusCodes**" is the HTTP status codes that indicate when an operation should be retried. The default values are HTTP status codes RequestTimeout(**408**), InternalServerError(**500**), BadGateway(**502**), ServiceUnavailable(**503**), GatewayTimeout(**504**).

FileReader Section

The "**FileReader**" section provides configuration settings that enable you to specify the capture file reader to use.

```
"FileReader" : {
   "CaptureDirectory" : "<capture-directory-path>",
   "UndeliverableDirectory" : "<undeliverable-directory-path>",
   "Delay" : <delay-in-milliseconds>,
   "RunContinuously" : <true-or-false>
}
```

The "**FileReader**" section parameters are listed in the following table.

Parameter	Description
CaptureDirectory	Directory path where capture files to be processed are located.
UndeliverableDirectory	Directory path where capture files that failed processing will be copied. This does not have to be on the same file system as the **CaptureDirectory**.
MaxRetries	Number of times to repeat a failed send to a Sender before giving up and copying the input file to the undeliverable directory.
Delay	Time in milliseconds before the Event Producer checks the capture directory for more files.
RunContinuously	Whether to continuously check for new files (**true** or **false**). If you select **false**, the Producer processes any files in the Capture Directory and then exits immediately.

FileTopicManager Section

The "**FileTopicManager**" section provides configuration for mapping Jade events to topics in Kafka or Hubs in Azure Event Hubs.

```
"FileTopicManager" : {
  "TopicFileName" : "<topic-config-file-path>"
}
```

The **TopicFileName** parameter specifies the full path to the JSON file containing topic configurations.

For more details about topic configuration, see "Topic Configuration" elsewhere in this chapter.

FileStorageProvider Section

The "**FileStorageProvider**" section in the configuration JSON specifies settings for the File System Storage Provider. This provider allows the Event Stream Producer to store blob and slob properties with sizes above a configured threshold value in the local file system. (For details, see "SizeThresholds", later in this topic.)

```
{
   "FileStorageProvider" : {
      "RootDirectory" : "<path-to-root-directory>"
   }
}
```

The **RootDirectory** parameter is required. It specifies the root directory where the File System Storage Provider will store blob or slob data, and it has a **String** type; for example:

```
"D:\\event-egress\\LobData"
```

For file paths and local configurations, determine values based on your system set up and requirements.

FileStorageProvider Usage

The **RootDirectory** parameter should point to a directory where the application has read and write permissions. The File System Storage Provider will create subdirectories within this root directory to organize stored data.

Directory Structure

The **FileStorageProvider** creates a hierarchical structure within the **RootDirectory**, as follows.

```
RootDirectory\
├── YYYY-MM\
│   ├── TransactionID1\
│   │   ├── [EventID1]__PropertyName1.blob
│   │   └── [EventID1]__PropertyName2.slob
│   └── TransactionID2\
│       ├── [EventID2]__PropertyName1.blob
│       └── [EventID2]__PropertyName2.slob
└── YYYY-MM\
└── ...
```

In this structure:

- **YYYY-MM** is the year and month of the event timestamp
- **TransactionID** is the unique identifier for the transaction
- **EventID** is the unique identifier for the event
- **PropertyName** is the name of the property associated with the Line of Business (LOB) data
- The **.blob** extension is used for binary large object (blob) data
- The **.slob** extension is used for string large object (slob) data

Example Configuration

In the following configuration example, the File System Storage Provider will use **"D:\event-egress\LobData"** as the root directory for storing large object (lob) data.

```
{
   "FileStorageProvider" :{
      "RootDirectory" : "D:\\event-egress\\LobData"
   }
}
```

Notes

When using the **FileStorageProvider** section:

- Ensure that the specified **RootDirectory** exists, and that the producer has appropriate permissions to create subdirectories and write to this directory.
- The File System Storage Provider will automatically create the necessary subdirectories within the **RootDirectory** and will attempt to create **RootDirectory** if it does not exist.
- Consider disk space requirements when choosing the **RootDirectory** location, especially for high-volume systems.
- For production environments, it is recommended that you use a dedicated disk or partition for the **RootDirectory**, to prevent interference with other system operations.

SizeThresholds Section

The "**SizeThresholds**" section in the configuration enables you to set limits for when large binary (blob) and string (slob) objects should be stored externally instead of being included directly in the JSON output.

```
"SizeThresholds" : {
  "SlobThresholdBytes" : "30KB",
  "BlobThresholdBytes" : "100KB"
}
```

The **"SizeThresholds"** values are listed in the following table.

Value	The maximum size for...
SlobThresholdBytes	String (slob) objects before they are stored externally
BlobThresholdBytes	Binary (blob) objects before they are stored externally

Value Format

Values can be specified as a:

- Plain number, representing bytes
- String with a **KB** or **MB** suffix for kilobytes or megabytes, respectively

Examples of the value format are as follows.

- "30KB" (30 kilobytes)
- "1MB" (1 megabyte)
- 50000 (50,000 bytes)

Behavior

If a slob property exceeds the **SlobThresholdBytes** value, it will be stored externally, and a URI referencing the stored data will be included in the JSON event instead.

If a blob property exceeds the **BlobThresholdBytes** value, it will be stored externally, and a URI referencing the stored data will be included in the JSON event instead.

Properties below these thresholds are included directly in the JSON event (strings as escaped text and binary data as base64-encoded strings).

Valid units for **BlobThresholdBytes** and **SlobThresholdBytes** are **B** (bytes), **KB** (kilobytes), and **MB** (megabytes). These values are case-insensitive. Setting both thresholds to zero (**0**) causes all blob and slob properties to be stored externally, regardless of size.

Default Values

If not specified in the configuration:

- **SlobThresholdBytes** defaults to **32,768** bytes (32 KB)
- **BlobThresholdBytes** defaults to **65,536** bytes (64 KB)

Notes

When a property is stored externally, a new property with the **$<propertyName>_Uri** is added to the JSON event and contains the URI of the stored data.

Proper configuration of these thresholds can help balance between JSON payload size and the number of external storage operations, optimizing for your specific use case and network conditions.

Logger Section

The "**Logger**" section provides configuration settings that enable you to specify the full path to the directory where log files are written.

```
"Logger" : {
   "LogFileName" : "<log-file-path>",
   "MaxLevel" : "<log-level>",
   "Delay" : <delay-in-milliseconds>,
   "RotateSize" : <size-in-megabytes>,
   "RotateKeep" : <number>,
   "Sink" : "<File-or-Console>"
}
```

The "**Logger**" parameters are listed in the following table.

Parameter	Description
LogFileName	Full path to the directory where log files are written. This is ignored if the value of the **Sink** parameter is set to **"Console"**. The path can contain one or more usages of the template **"${ProducerName}"**, each of which will expand to the name of the current producer (which is set on the command line). **Note** This unique directory name cannot be shared if multiple **JadeEventProducer.exe** applications are run.
MaxLevel	Maximum log level (for example, **"Debug"**, **"Info"**, **"Warn"**, or **"Error"**).
Delay	Milliseconds before log messages are flushed to disk. Severity **warn** or above are always immediately flushed. If not set, it defaults to **1000** (1 second). It has no effect if the value of the **Sink** parameter is set to **"Console"**.
RotateSize	If set, log files will be rotated when reaching the specified size in megabytes. Setting to zero (**0**) disables log rotation. It has no effect if the value of the **Sink** parameter is set to **"Console"**.
RotateKeep	Number of old logs to keep before deletion. It is ignored if rotation is not enabled.
Sink	Medium to which to log. Valid options are **"File"** to log the file specified by **LogFileName** or **"Console"** to log to the **stdout** file stream. The default value is **"File"**.

The following is an example of the **Logger** section.

```
"Logger" : {
    "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Info.log",
    "MaxLevel" : "Info"
}
```

Loggers Section

The "**Loggers**" section provides configuration settings that enable you to specify one or more **Logger** sections; for example, for logging to both a **"File"** and **"Console"**, or to have different files for different log levels.

```
"Loggers" : [
   {
      "LogFileName" : "<log-file-path>",
      "MaxLevel" : "<log-level>",
      "Delay" : <delay-in-milliseconds>,
      "RotateSize" : <size-in-megabytes>,
      "RotateKeep" : <number>,
      "Sink" : "<File-or-Console>"
   },
   ...
]
```

"**Loggers**" is an array of "**Logger**" configurations, with values that match those listed in the **Logger** section table.

Note Only one of the "**Logger**" or "**Loggers**" sections can be present.

The effective **Delay** setting will be the lowest configured value, or the default setting if no configuration specifies a **Delay** value.

The following is an example of the **Loggers** section.

```
"Loggers" :
      [
         {
            "MaxLevel" : "Debug",
            "Sink" : "File",
            "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Debug.log"
         },
         {
            "MaxLevel" : "Info",
            "Sink" : "File",
            "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Info.log"
         },
         {
            "MaxLevel" : "Info",
            "Sink" : "Console"
         }
      ]
```

HealthMonitor Section

To send regular health status information for the producer component of Jade Event Streaming, include the **HealthMonitor** section in the configuration. For details about the format of the output health monitoring message, see "Health Monitoring Data Specification", in Chapter 4.

The "**HealthMonitor**" section provides configuration settings that enable you to run the Health Monitor for a producer.

```
"HealthMonitor" :
   {
      "Heartbeat" : <period-in-seconds>,
      "SenderStalledTimeout" : <period-in-seconds>, // AzureSenders only
      "AdditionalFields" : [
```

```
            "ProducerStarted",
            "LastEventSent",
            "EventsSentToDeadLetter",
            "ErrorsInPeriod",
            "EventsInPeriod",
            "TotalSenders", // AzureSenders only
            "AliveSenders" // AzureSenders only
        ],
        "Format" : {
            "AzureMonitor" : <true-or-false>,
            "CamelCasedPropertyNames" : <true-or-false>
        },
        "RestPublisher" : {
            "Endpoint" : "<https://<namespace>",
            "Authentication" : { "Type":"Authentication-type" },
            "RetryOptions":{
               "MaxRetries" : <number-of-retries>,
               "RetryDelay" : <period-in-milliseconds>,
               "MaxRetryDelay" : <period-in-milliseconds>,
               "StatusCodes" : [<number>, <number, ...>]
             },
            "HttpLogging" : <true-or-false>
       },
       "TopicPublisher": {
           "Topic" : "<topic-name>"
       }
    }
```

The "**HealthMonitor**" properties are listed in the following table.

Property	Description
Heartbeat	The cadence with which the health monitoring message is sent. This is a positive integer value specified in seconds. The default value is **60** seconds; the minimum value is **10** seconds.
SenderStalledTimeout	If present, this setting specifies the timeout period (in seconds) after which an unresponsive Sender is deemed to have stalled. The default period is **60** seconds and the minimum period is **30** seconds. Set this value to zero (**0**) to disable this check. If you specify a **SenderStalledTimeout** section, include the **TotalSenders** and **AliveSenders** options in the **AdditionalFields** section. **Note** This section applies to AzureSenders only.

Property	Description
AdditionalFields	An array of field names that are the names of the data properties in the health monitoring message. It is always assumed that **Producer** and **TimeGenerated** will be in the message. These are emitted even if the **AdditionalFields** property is not present. The additional field options are: ■ "**ProducerStarted**", which is the time the producer was started. ■ "**LastEventSent**", which is the time of the last health monitoring message for a period that contained any events. ■ "**EventsSentToDeadLetter**", which is the number of events that have been sent to the **FileReaderUndeliverableDirectory** in the lifetime of this Producer run. ■ "**ErrorsInPeriod**", which is a count of any errors that have occurred while sending events to the broker since the last health message. ■ "**EventsInPeriod**", which is the count of event sends to all topics since the last health message. If you have included a **SenderStalledTimeout** setting, specify the following. ■ "**TotalSenders**", which is the count of the total number of senders. ■ "**AliveSenders**", which is the count of responsive senders. The **TotalSenders** and **AliveSenders** counts should be equal to and have the same value as the **NumberOfSenders** parameter in the **AzureSender** section. If a sender stops responding, the **AliveSenders** count will reduce. The number of stalled senders is logged in an info-level health monitor message. **Note** The **TotalSenders** and **AliveSenders** options apply to AzureSenders only.
Format	If present, this section allows the user to modify the packet from the standard schema. It has the following fields. ■ **AzureMonitor**. If present and **true**, the health monitoring message is inserted into a JSON array (which will then have the health monitoring message as its only member). This complies with the Azure Monitor data requirements. ■ **CamelCasedPropertyNames**. If present and **true**, the property names will switch from Pascal case to Camel case (that is, the property names will have a lowercase first character.) This could be useful for health consumers written in Jade.
RestPublisher	If present, this is an object that specifies a REST endpoint to which the health monitoring message is sent. It has the following fields. ■ **Endpoint**, which specifies the URL endpoint to which the health message is sent.

Property	Description
	■ **Authentication**. The authentication **Type** can be one of the following. ▫ **AzureLogin** ▫ **AzureClientSecret** ▫ **AzureEnvironment** ▫ **AzureManagedIdentity** ▫ **None** For details, see "Type-Specific Parameters", later in this chapter. ■ **RetryOptions**, which optionally configures the retry behavior of the REST connection. It allows fine-tuning of the retry mechanism to make the availability of the service more resilient. Even if you do not specify a **RetryOptions** value, the retry will occur using the default values in the following list. The retry options are: ▫ "**MaxRetries**", which is the maximum number of retry attempts before reporting an error. The default value is **3**. ▫ "**RetryDelay**", which is the minimum delay in milliseconds between retry attempts. The default value is **800** milliseconds. ▫ "**MaxRetryDelay**", which is the maximum delay in milliseconds between retry attempts. The default value is **60** seconds. ▫ "**StatusCodes**", which is an array of status codes as Integers that indicate when an operation should be retried. The default values are HTTP status codes **408** (RequestTimeout), **500** (InternalServerError), **502** (BadGateway), **503** (ServiceUnavailable), **504** (GatewayTimeout). If you specify status codes, the default codes are not included; that is, if you do not specify any codes, you will get **408**, **500**, **502**, **503**, and **504**. If you specify **[500, 601]**, you will only get **500** and **601** (and not **408**, **502**, **503**, or **504**). ■ **HttpLogging**, which is a Boolean value that indicates whether Azure internal HTTP logging should be logged or reported for this connection. This parameter defaults to a value of **false** if unspecified. If **true**, logging will be sent to the logger specified in the "**Logger**" section of the configuration file and log-level filtering will be performed based on the level of that log. (This logging can be quite verbose, so it is recommended to turn it on only for debug scenarios.)
TopicPublisher	If present, the health monitoring message will be produced to a topic in the current broker (either an Azure EventHub or a Kafka topic). The **Topic** parameter specifies the name of the topic to which to send the health monitoring message.

When the configuration file is read on start up, if there is no **HealthMonitor** section specified or it contains an error, the error is logged and the producer will continue with no health monitoring.

Malformed or otherwise incorrect **RestPublisher** property **Endpoint** or **TopicPublisher** property **Topic** values are detected when the first message is sent. An error is logged and the producer will continue unmonitored.

The Health Monitor runs on a separate thread from the **Reader** and **Senders**. It will wake up with the cadence of the **Heatbeat**, construct the message, and then send it to the configured location. If an error occurs in the connection or sending of the message, the Health Monitor will log an error and operation will continue as normal on the next heartbeat. If an unexpected error occurs, health monitoring will cease and the producer will continue unmonitored.

It is expected that any external agents that are monitoring these messages will detect the absence of the Health Monitor. The producer will have to be restarted for health monitoring to be restored.

The following is an example of the **HealthMonitor** section.

```
"HealthMonitor" :
   {
      "Heartbeat" : 10,
      "SenderStalledTimeout" : 60,
      "AdditionalFields" : [
          "ProducerStarted",
          "LastEventSent",
          "EventsSentToDeadLetter",
          "ErrorsInPeriod",
          "EventsInPeriod",
          "TotalSenders",
          "AliveSenders"
      ],
      "Format" : {
          "AzureMonitor" : true,
          "CamelCasedPropertyNames" : false
      },
      "RestPublisher" : {
          "Endpoint" : "https://monitor.company.co.nz/monitor",
          "RetryOptions":{
              "MaxRetries" : 5,
              "RetryDelay" : 1000,
              "MaxRetryDelay" : 30,
              "MaxRetryDelay" : 30,
              "StatusCodes" : [408, 500, 502, 503, 504, 403]
           },
          "Authentication" : { "Type" : "None" },
          "HttpLogging" : false
     },
     "TopicPublisher": {
         "Topic" : "my-eventhub-healthevents"
     }
}
```

Applies to Version: 2025.0.01 and higher

Event Stream Producer Configuration Examples

The following are examples of Event Stream Producer (ESP) configurations.

- Development and Testing Configuration with FileSender Example
- File Storage Provider Configuration Example
- Kafka Sender Configuration Example
- Event Hub with Azure Schema Registry and File Blob Storage Configuration Example
- Event Hub with Azure Schema Registry and Azure Blob Storage Configuration Example

- Azure Event Hubs Configuration with Azure Schema Registry and Azure Blob Storage Example
- Confluent Cloud Configuration with Schema Registry Example

Development and Testing Configuration with FileSender Example

This configuration example is designed for local development and testing. It uses a **FileSender** to write events to the local file system instead of to a live messaging service. It includes a single file-based logger for detailed debugging and sets up local directories for file reading, topic management, and blob storage.

This development and testing configuration with **FileSender** setup allows developers to try out the ESP without needing access to cloud resources.

```
{
   "Modules" : {
      "Reader" : "File",
      "Sender" : "File",
      "TopicManager" : "File",
      "Serializer" : "Json",
      "StorageProvider" : "File",
      "SchemaGenerator" : "Json",
      "SchemaRegistry" : "None"
   },
   "Configurations" : {
      "FileReader" : {
         "CaptureDirectory" : "C:\\Events\\ChangeCapture",
         "UndeliverableDirectory" : "C:\\Events\\ChangeCapture\\undeliverable",
         "RunContinuously" : true,
         "Delay" : 200
      },
      "FileTopicManager" : {
         "TopicFileName" : "C:\\Events\\Config\\JadeTopics.config.json"
      },
      "FileSender" : {
         "OutputDirectory" : "C:\\events\\EventCapture"
      },
      "FileStorageProvider" : {
         "RootDirectory" : "C:\\Events\\ChangeCapture\\Blobs"
      },
      "SizeThresholds" : {
         "SlobThresholdBytes" : "300KB",
         "BlobThresholdBytes" : "300KB"
      },
      "Catalog" : {
         "CatalogDirectory" : "C:\\Jade\\System\\Journals"
      },
      "SchemaCache" : {
         "CacheRootDirectory" : "C:\\Events\\SchemaCache"
      },
      "Logger" : {
         "LogFileName" : "C:\\Events\\Logs\\${producername}_producer.log",
         "MaxLevel" : "Debug",
         "Delay" : 1000,
         "RotateSize" : 10,
         "RotateKeep" : 5,
         "Sink" : "File"
```

```
        }
    }
}
```

Key aspects of this development and testing configuration with **FileSender** are as follows.

- Uses **FileSender** for local file output, suitable for development and testing.
- Single file **Logger** is configured for detailed debugging.
- **FileReader** is set to run continuously.
- All directories are set to local paths for easy access and monitoring.

File Storage Provider Configuration Example

This configuration example uses the **File** storage provider.

```
{
   "Modules" : {
      "Reader" : "File",
      "Sender" : "File",
      "TopicManager" : "File",
      "Serializer" : "Json",
      "StorageProvider" : "File",
      "SchemaGenerator" : "Json",
      "SchemaRegistry" : "None"
   },
   "Configurations" : {
      "FileReader" : {
         "CaptureDirectory" : "C:\\Events\\ChangeCapture",
         "UndeliverableDirectory" : "C:\\Events\\ChangeCapture\\undeliverable",
         "RunContinuously" : true,
         "Delay" : 200
      },
      "FileTopicManager" : {
         "TopicFileName" : "C:\\Events\\Config\\JadeTopics.config.json"
      },
      "FileSender" : {
         "OutputDirectory" : "t:\\autoqual\\server\\EventCapture"
      },
      "FileStorageProvider" : {
         "RootDirectory" : "C:\\Events\\ChangeCapture\\Blobs"
      },
      "SizeThresholds" : {
         "SlobThresholdBytes" : "300KB",
         "BlobThresholdBytes" : "300KB"
      },
      "Catalog" : {
         "CatalogDirectory" : "C:\\Jade\\System\\Journals"
      },
      "SchemaCache" : {
         "CacheRootDirectory" : "C:\\Events\\SchemaCache"
      },
      "Logger" : {
         "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Info.log"
```

```
            "MaxLevel" : "Info"
         }
      }
   }
```

Kafka Sender Configuration Example

This production-ready configuration example uses the **Kafka** sender and **File** storage provider.

```
{
   "Modules" : {
      "Reader" : "File",
      "Sender" : "Kafka",
      "TopicManager" : "File",
      "Serializer" : "Json",
      "StorageProvider" : "File",
      "SchemaGenerator" : "Json",
      "SchemaRegistry" : "None"
   },
   "Configurations" : {
      "FileReader" : {
         "CaptureDirectory" : "C:\\Events\\ChangeCapture",
         "UndeliverableDirectory" : "C:\\Events\\ChangeCapture\\undeliverable",
         "RunContinuously" : true,
         "Delay" : 200
      },
      "FileTopicManager" : {
         "TopicFileName" : "C:\\Events\\Config\\JadeTopics.config.json"
      },
      "KafkaSender" : {
         "Properties" : {
            "bootstrap.servers" : "kafkahostmachine:9094",
            "security.protocol" : "plaintext"
         }
      },
      "FileStorageProvider" : {
         "RootDirectory" : "C:\\Events\\ChangeCapture\\Blobs"
      },
      "SizeThresholds" : {
         "SlobThresholdBytes" : "300KB",
         "BlobThresholdBytes" : "300KB"
      },
      "Catalog" : {
         "CatalogDirectory" : "C:\\Jade\\System\\Journals"
      },
      "SchemaCache" : {
         "CacheRootDirectory" : "C:\\Events\\SchemaCache"
      },
      "Logger" : {
         "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Info.log",
         "MaxLevel" : "Info"
      }
   }
}
```

Event Hub with Azure Schema Registry and File Blob Storage Configuration Example

This production-ready configuration example uses Azure Kafka Sender, Azure Schema Registry, and File Storage Provider.

```
{
   "Modules" : {
      "Reader" : "File",
      "Sender" : "Kafka",
      "TopicManager" : "File",
      "Serializer" : "Json",
      "StorageProvider" : "File",
      "SchemaGenerator" : "Json",
      "SchemaRegistry" : "Azure"
   },
   "Configurations" : {
      "FileReader" : {
         "CaptureDirectory" : "C:\\Events\\ChangeCapture",
         "UndeliverableDirectory" : "C:\\Events\\ChangeCapture\\undeliverable",
         "RunContinuously" : true,
         "Delay" : 200
      },
      "FileTopicManager" : {
         "TopicFileName" : "C:\\Events\\Config\\JadeTopics.config.json"
      },
      "KafkaSender" : {
         "Properties" : {
            "bootstrap.servers" :
                  "<your-eventhub-namespace>.servicebus.windows.net:9093",
            "security.protocol" : "SASL_SSL",
            "sasl.mechanisms" : "PLAIN",
            "sasl.username" : "$ConnectionString",
            "sasl.password" : "Endpoint=sb://<your-eventhub-namespace>.
                 servicebus.windows.net/;
                 SharedAccessKeyName=keyname;SharedAccessKey=key"
         }
      },
      "FileStorageProvider" : {
         "RootDirectory" : "C:\\Events\\ChangeCapture\\Blobs"
      },
      "SizeThresholds" : {
         "SlobThresholdBytes" : "300KB",
         "BlobThresholdBytes" : "300KB"
      },
      "AzureSchemaRegistry" : {
         "Endpoint" : "https://<your-eventhub-namespace>.servicebus.windows.net",
         "EventhubNamespace" : "<your-eventhub-namespace>",
         "SchemaGroup" : "<your-eventhub-json-schema-group>",
         "SchemaFormat" : "Json"
         }
      },
      "Catalog" : {
         "CatalogDirectory" : "C:\\Jade\\System\\Journals"
      },
```

```
        "SchemaCache" : {
           "CacheRootDirectory" : "C:\\Events\\SchemaCache"
        },
        "Loggers" :
        [
           {
              "MaxLevel" : "Debug",
              "Sink" : "File",
              "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Debug.log"
           },
           {
              "MaxLevel" : "Info",
              "Sink" : "File",
              "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Info.log"
           },
           {
              "MaxLevel" : "Info",
              "Sink" : "Console"
           }
        ]
     }
}
```

Event Hub with Azure Schema Registry and Azure Blob Storage Configuration Example

This production-ready configuration example targets Azure. It uses Azure Kafka Sender, Azure Schema Registry, and Azure Blob Storage Provider.

This Azure configuration example uses **AzureBlobStorageProvider** for large blob and slob data. It includes both file and console logging for monitoring in a production environment.

```
{
   "Modules" : {
      "Reader" : "File",
      "Sender" : "Kafka",
      "TopicManager" : "File",
      "Serializer" : "Json",
      "StorageProvider" : "AzureBlobStorage",
      "SchemaGenerator" : "Json",
      "SchemaRegistry" : "Azure"
   },
   "Configurations" : {
      "FileReader" : {
      "CaptureDirectory" : "C:\\Events\\ChangeCapture",
      "UndeliverableDirectory" : "C:\\Events\\ChangeCapture\\undeliverable",
      "RunContinuously" : true,
      "Delay" : 200
      },
      "FileTopicManager" : {
         "TopicFileName" : "C:\\Events\\Config\\JadeTopics.config.json"
      },
      "KafkaSender" : {
         "Properties" : {
            "bootstrap.servers" :
                  "<your-eventhub-namespace>.servicebus.windows.net:9093",
```

```
      "security.protocol" : "SASL_SSL",
      "sasl.mechanisms" : "PLAIN",
      "sasl.username" : "$ConnectionString",
      "sasl.password" : "Endpoint=sb://<your-eventhub-namespace>.
            servicebus.windows.net/;SharedAccessKeyName=keyname;
            SharedAccessKey=key"
   }
},
"AzureBlobStorageProvider" : {
   "ContainerName" : "<blob-container-name>",
   "StagingDirectory" : "C:\\Events\\BlobStaging",
   "BlobPrefix" : "events",
   "Authentication" : {
      "Type : "ConnectionString",
      "ConnectionString" : "DefaultEndpointsProtocol=https;
                            AccountName=<storage-account-name>;
                            AccountKey=<account-key>;
                            EndpointSuffix=core.windows.net"
   }
},
"SizeThresholds" : {
   "SlobThresholdBytes" : "300KB",
   "BlobThresholdBytes" : "300KB"
},
"AzureSchemaRegistry" : {
   "Endpoint" : "https://<your-eventhub-namespace>.servicebus.windows.net",
   "EventhubNamespace" : "<your-eventhub-namespace>",
   "SchemaGroup" : "<your-eventhub-json-schema-group>",
   "SchemaFormat" : "Json",
   "Authentication" : {
      "Type" : "AzureLogin"
   }
},
"Catalog" : {
   "CatalogDirectory" : "C:\\Jade\\System\\Journals"
},
"SchemaCache" : {
   "CacheRootDirectory" : "C:\\Events\\SchemaCache"
},
"Loggers" :
[
   {
      "MaxLevel" : "Debug",
      "Sink" : "File",
      "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Debug.log"
   },
   {
      "MaxLevel" : "Info",
      "Sink" : "File",
      "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Info.log"
   },
   {
      "MaxLevel" : "Info",
      "Sink" : "Console"
   }
]
```

```
      }
   }
```

Azure Event Hubs Configuration with Azure Schema Registry and Azure Blob Storage Example

This production-ready configuration example is tailored for Microsoft Azure services. It uses the **EventHubSender** to integrate with Azure Event Hubs for event streaming and includes Azure Schema Registry for schema management. Large blob and slob data are stored in Azure Blob Storage.

This configuration example includes both file and console logging, suitable for production monitoring.

```
{
   "Configurations" : {
      "AzureSender" : {
         "Authentication" : {
           "Type" : "ConnectionString",
           "ConnectionString" :
              "Endpoint=sb://<your-eventhub-namespace>.servicebus.windows.net/;
               SharedAccessKeyName=your-shared-access-policy;
               SharedAccessKey=your-key",
          }
       },
      "AzureSchemaRegistry" : {
         "Endpoint" : "https://<your-schema-registry>.servicebus.windows.net",
         "EventhubNamespace" : "<your-eventhub-namespace>",
         "SchemaGroup" : "<your-eventhub-json-schema-group>",
         "SchemaFormat" : "Json",
         "Authentication" : {
           "Type" : "AzureClientSecret",
           "ClientId" : "<your-client-id>",
           "ClientSecret" : "<your-client-secret>",
           "TenantId" : "<your-tenant-id>"
         }
      },
      "FileReader" : {
         "CaptureDirectory" : "c:\\events\\capture-files",
         "Delay" : 1000,
         "RunContinuously" : true,
         "UndeliverableDirectory" : "c:\\events\\undeliverable"
      },
      "FileTopicManager" : {
         "TopicFileName" : "c:\\events\\config\\topic.config.json"
      },
      "Loggers" : [
         {
            "Delay" : 1000,
            "LogFileName" : "c:\\events\\logs\\event-producer-one.log",
            "MaxLevel" : "Info",
            "RotateKeep" : 10,
            "RotateSize" : 100,
            "Sink" : "File"
         },
         {
            "MaxLevel" : "Warn",
            "Sink" : "Console"
```

```
            }
         ],
         "AzureBlobStorageProvider" : {
            "Endpoint" : "https://<your-storage-account-name>.blob.core.windows.net,
            "BlobPrefix" : "events",
            "ContainerName" : "<event-lob-data>",
            "StagingDirectory" : "<staging-root-directory>",
            "Authentication" : {
               "Type" : "AzureLogin"
             }
          },
         "SizeThresholds" : {
            "BlobThresholdBytes" : "5MB",
            "SlobThresholdBytes" : "1MB"
         },
         "Catalog" : {
            "CatalogDirectory" : "C:\\Jade\\System\\Journals"
         },
         "SchemaCache" : {
           "CacheRootDirectory" : "C:\\Events\\SchemaCache"
         }
      },
      "Modules" : {
         "Reader" : "File",
         "Sender" : "Azure",
         "Serializer" : "Json",
         "StorageProvider" : "AzureBlobStorage",
         "TopicManager" : "File",
         "SchemaGenerator" : "Json",
         "SchemaRegistry" : "Azure"
      }
   }
```

Key aspects of this Azure Event Hubs with Azure Schema Registry and Azure Blob Storage configuration example are listed in the following table.

Element	Description
AzureSender	Configured for Azure Event Hubs
AzureSchemaRegistry	Includes Azure Schema Registry configuration
Loggers	Contains two configurations: one for detailed file logging and another for console warnings
AzureBlobStorageProvider	Used for storing large blobs and slobs in Azure Blob Storage
SizeThresholds	Set to determine when to use external storage
Authentication	Contains two authentication types for each of the Azure services (for demonstration purposes only)

Confluent Cloud with Schema Registry Configuration Example

This production-ready configuration example targets Confluent Cloud, a fully managed Apache Kafka service. It uses the **KafkaSender** with specific security settings for Confluent Cloud and includes Schema Registry integration.

This Confluent schema registry configuration example uses a local **FileStorageProvider** for large blob and slob data, demonstrating a hybrid approach with cloud messaging and local storage. It includes file and console logging for monitoring in a production environment.

```
{
   "Modules" : {
      "Reader" : "File",
      "Sender" : "Kafka",
      "TopicManager" : "File",
      "Serializer" : "Json",
      "StorageProvider" : "File",
      "SchemaGenerator" : "Json",
      "SchemaRegistry" : "Confluent"
   },
   "Configurations" : {
      "KafkaSender" : {
         "Properties" : {
            "bootstrap.servers" : "your-kafka-server.confluent.cloud:9092",
            "security.protocol" : "SASL_SSL",
            "sasl.mechanisms" : "PLAIN",
            "sasl.username" : "<API_KEY>",
            "sasl.password" : "<API_SECRET>",
            "schema.registry.url" :
                 "https://your-schema-registry.confluent.cloud",
            "schema.registry.basic.auth.user.info" : "<SR_API_KEY>:
                 <SR_API_SECRET>",
            "basic.auth.credentials.source" : "USER_INFO"
         }
      },
      "Loggers" : [
         {
            "LogFileName" : "C:\\Events\\Logs\\JadeEventProducer_Info.log",
            "MaxLevel" : "Info",
            "Delay" : 1000,
            "RotateSize" : 100,
            "RotateKeep" : 10,
            "Sink" : "File"
         },
         {
            "MaxLevel" : "Warn",
            "Sink" : "Console"
         }
      ],
      "FileReader" : {
         "CaptureDirectory" : "C:\\Events\\ChangeCapture",
         "UndeliverableDirectory" : "C:\\Events\\ChangeCapture\\undeliverable",
         "Delay" : 1000,
         "RunContinuously" : true
      },
      "FileTopicManager" : {
         "TopicFileName" : "C:\\Events\\Config\\JadeTopics.config.json"
```

```
        },
        "FileStorageProvider" : {
           "RootDirectory" : "C:\\Events\\ChangeCapture\\Blobs"
        },
        "SizeThresholds" : {
           "SlobThresholdBytes" : "1MB",
           "BlobThresholdBytes" : "5MB"
        },
        "ConfluentSchemaRegistry" : {
           "Endpoint" : "https://endpoint.confluent.cloud",
           "SchemaFormat" : "Json",
           "Authentication" : {
             "Type" : "ApiKey",
             "ApiKeyName" : "1234567890",
             "ApiKeyValue" : "ABCDEFEGGHIJKLMNOPQRSTUVWXYZ1234567890+"
           }
        },
        "Catalog" : {
           "CatalogDirectory" : "C:\\Jade\\System\\Journals"
        },
        "SchemaCache" : {
           "CacheRootDirectory" : "C:\\Events\\SchemaCache"
        }
     }
  }
```

Key aspects of this Confluent **SchemaRegistry** configuration **FileSender** example are listed in the following table.

Element	Description
KafkaSender	Configured for Confluent Cloud with security settings
ConfluentSchemaRegistry	Includes configuration for Confluent Cloud
Loggers	Contains a configuration for two **Loggers**: one for detailed file logging and another for console warnings
FileStorageProvider	Used for storing large blobs and slobs locally
SizeThresholds	Set to determine when to use external storage

OAuth2 on Confluent Cloud with Schema Registry

The following example shows the use of the **ConfluentOAuth2** authentication type with a SchemaRegistry.

```
  "ConfluentSchemaRegistry": {
           "Authentication":{
             "Type": "ConfluentOAuth2",
             "TenantId": "my-tenant-id",
             "ClientId": "my-client-id",
             "ClientSecret": "my-client-secret",
             "Scope": "<https://cluster-id>.<region>.confluent.cloud/>",
             "IdentityPool": "pool-xxxxx",
             "TargetCluster" : "lsrc-yyyyyy"
  }
```

The **IdentityPool** and **TargetCluster** parameters are required to use the Confluent **SchemaRegistry** with OAuth2. These parameters configure the Confluent Cloud SchemaRegistry to use the associated Azure Active Directory (AD) identity provider. For details, see "Authentication Types", later in this chapter.

Topic Configuration

This topic explains the configuration for mapping Jade events to topics in Kafka or Hubs in Azure Event Hubs.

Configuration Structure

```
{
  "TopicName1" : [ ... ],
  "TopicName2" : [ ... ],
...
}
```

Each topic or hub is represented by an array of mapping rules.

Mapping Rules

The mapping rules are as follows.

- Class-based mapping
- Event-based mapping

Class-Based Mapping

The class-based mapping rule sends all events for the specified class and (optionally) its subclasses to the topic.

```
{
   "Class" : "SchemaName::ClassName"
}
```

The class-based mapping values are listed in the following table.

Value	Name of the...
SchemaName	Schema in Jade
ClassName	Class in Jade

Event-Based Mapping

The event-based mapping rule sends all events of the specified type to the topic, regardless of the class.

```
{
  "*" : "EventName"
}
```

The wildcard "*****" value indicates all classes.

The "**EventName**" value specifies the name of the event type. Only **"TransactionCommitted"** is supported at this time.

Special Wildcard Mapping

The special wildcard mapping rule sends events for all classes to the topic.

```
{
  "Class" : "*"
}
```

This rule sends events for all classes to the topic.

Example Configuration

The following is an example of topic configuration.

```
{
    "OrgEvents" : [
        { "Class" : "BankModelSchema::Bank" },
        { "Class" : "BankModelSchema::District" }
    ],
    "AccountEvents" : [
        { "Class" : "BankModelSchema::Account" }
    ],
    "CustomerEvents" : [
        { "Class" : "BankModelSchema::Customer" }
    ],
    "TransactionEvents" : [
        { "Class" : "BankModelSchema::Transaction" }
    ],
    "TransactionCommitEvents" : [
        { "*" : "TransactionCommitted" }
    ],
    "JadeAllEvents" : [
        { "Class" : "*" },
        { "*" : "TransactionCommitted" }
    ]
}
```

Configuration Breakdown

The topic configuration elements are listed in the following table.

Event	Receives...
OrgEvents	Events for **BankModelSchema::Bank** and **BankModelSchema::District** classes.
AccountEvents	Events for the **BankModelSchema::Account**.
CustomerEvents	Events for the **BankModelSchema::Customer** class.
TransactionEvents	Events for the **BankModelSchema::Transaction** class.
TransactionCommitEvents	All **TransactionCommitted** events, regardless of the class.
JadeAllEvents	Events for all classes and all **TransactionCommitted** events. This topic will receive a copy of all events sent to other topics.

Best Practices

Topic configuration best practices are as follows.

- Use specific class mappings for granular control over event routing.
- Use event-based mappings for cross-cutting concerns like transaction commits.
- Ensure that topic names in this configuration match the topics in your Kafka cluster or Event hubs in your Azure Event Hubs namespace.
- Review and update this configuration when new classes are added to your persistent data model.

Authentication Configuration

This section describes the authentication configuration used to connect to the **Schema Registry**, **EventHubs**, **Blob Storage**, and **HealthMonitor** services.

The **Authentication** element has several settings, with a mandatory **Type** specifying the kind of credentials to use, along with additional type-specific values.

Note Authentication configuration for Event Streaming requires up to three different connections (for example, Azure Event Hub, Azure Blob Storage, and Azure Schema Registry) to be correctly set up for proper functionality.

Mis-configurations can cause connection failures so it is essential to carefully verify credentials, permissions, and connection settings. Ensure all required authentication parameters are correctly configured, considering HTTPS enforcement, proxy settings, and firewall rules to avoid connectivity issues and unnecessary troubleshooting. This applies to configuration file settings and to cloud-based portal configurations.

Authentication Types

The **Type** for the **Authentication** parameter can be one of the following.

Value	Description
AzureLogin	Uses **AzureCliCredentials**, requiring the **az login** command to be run on the host machine. **Note** **AzureLogin** requires the **Azure Event Hub Data Owner** or **Azure Event Hubs Data Sender** role to be assigned to the **Namespace** for the user account running **az login**.
AzureClientSecret	Uses **AzureClientSecretCredentials**.
AzureEnvironment	Uses **AzureEnvironmentCredentials**, which relies on **AZURE_*xxx*** environment variables.
AzureManagedIdentity	Uses **AzureManagedIdentityCredentials**.
ConnectionString	Alternative combination of Endpoint and credentials, which is often used in a development scenario. This type is not available for Schema Registry.
SharedAccessKey	For Azure access, uses the key from a Shared Access Policy.
ApiKey	Useful for Confluent development purposes (for Schema Registry only).
ConfluentOAuth2	Specify this authentication type to use OAuth2 with a Confluent Cloud environment.
None	There are no additional values to specify.

For details, see "Type-Specific Parameters", in the following section.

Type-Specific Parameters

Some authentication types require additional parameters, as follows.

AzureClientSecret

For **AzureClientSecret**, you must provide the following credential values for your Azure App registration.

Credential	Description
TenantId	Azure Active Directory (AD) tenant ID associated with your subscription.
ClientId	Client ID of an Azure AD application that has access to the **Schema Registry**, **Storage Provider**, or **Event Hubs Namespace**.
ClientSecret	Client secret for the Azure AD application specified by the **ClientId** value.

Example Configuration

The following is an example "**Authentication**" configuration for the **AzureClientSecret** type.

```
"Authentication" : {
   "Type" : "AzureClientSecret",
   "TenantId" : "99998888-ffff-7777-eeee-6666dddd5555",
   "ClientId" : "00001111-aaaa-2222-bbbb-3333cccc4444",
   "ClientSecret" : "abcdefghijklmnopqrstuvwxyz0123456789/+ABCDEFGHIJKLMNOP==" }
```

AzureEnvironment

For **AzureEnvironment**, the same credential values as those in the previous table under "**AzureClientSecret**" are used but they are read from fixed environment variables, as follows.

Fixed Environment Variable	Description
AZURE_TENANT_ID	Azure Active Directory (AD) tenant ID.
AZURE_CLIENT_ID	Client ID of the Azure AD application.
AZURE_CLIENT_SECRET	Client secret for the Azure AD application.

AzureManagedIdentity

For **AzureManagedIdentity**, no additional credential values are required in the configuration file but it *does* require configuring in Azure.

When the Event Streaming Producer is hosted on an Azure Windows virtual machine (VM), **AzureManagedIdentity** authentication is available to an Event Hubs environment. For more details, see the following URL.

https://learn.microsoft.com/en-us/azure/event-hubs/authenticate-managed-identity

Event Hub and Schema Registry Requirements

Add the **Event Hubs Data Owner** role to Azure VM System Identity for the ***<example-eventhub-namespace>*** Event Hubs namespace.

Azure Blob Storage Requirements

Add the **Storage Blob Data Contributor** role to Azure VM System Identity for the ***<example-blob-storage>*** Storage account.

ConnectionString

For **ConnectionString**, you must provide the Connection String value.

Example Configuration

The following is an example "**Authentication**" configuration for the **ConnectionString** type.

```
"Authentication" : {
   "Type" : "ConnectionString",
   "ConnectionString" : "Endpoint=sb://myeventhubs.servicebus.windows.net/;
                         SharedAccessKeyName=RootManageSharedAccessKey;
                         SharedAccessKey=abc123..." }
```

SharedAccessKey

For **SharedAccessKey**, you must provide the values listed in the following table.

Value	Description
PolicyName	The name you have created in Azure for the Shared Access Policy.
Key	The primary or secondary key generated by Azure for this policy.

Note **SharedAccessKey** authentication is not supported for **AzureSchemaRegistry**.

Example Configuration

The following is an example "**Authentication**" configuration for the **SharedAccessKey** type.

```
"Authentication" : {
   "Type" : "SharedAccessKey",
   "PolicyName" : "MyEventHubPolicy",
   "Key" : "abc123" }
```

ApiKey

For **ApiKey**, you must provide the values listed in the following table.

Value	Description
ApiKeyName	API key name or ID.
ApiKeyValue	Secret associated with the API key.

This type is primarily used for development purposes in a Confluent Cloud Kafka environment and is not recommended for production environments.

Example Configuration

The following is an example "**Authentication**" configuration for the **ApiKey** type.

```
"Authentication" : {
   "Type" : "ApiKey",
   "ApiKeyName" : "1234567890",
   "ApiKeyValue" : "ABCDEFEGGHIJKLMNOPQRSTUVWXYZ1234567890+"}
```

ConfluentOAuth2

The **ConfluentOAuth2** type using an Azure Active Directory (AD) identity provider uses similar settings to the **AzureClientSecret** along with three additional credentials as listed in the following table.

Credential	Description
TenantId	Azure AD tenant ID associated with your subscription.
ClientId	Client ID of an Azure AD application that has access to the **Schema Registry**.
ClientSecret	Client secret for the Azure AD application specified by the **ClientId** value.
Scope	Specifies the extent or context of the Authentication request. For Confluent, specify the URL of the Confluent environment.
IdentityPool	Specifies which identity pool is associated with the SchemaRegistry. This must have been configured in Confluent and associated with the Azure-based identity provider being used. This setting is required only if using the Confluent SchemaRegistry.
TargetCluster	Specifies the SchemaRegistry Cluster that is configured to be used on Confluent. Note that this is different from the Kafka cluster. This setting is required only if using the Confluent SchemaRegistry.

Contact Jade Support for additional guidelines about setting up the **ConfluentOAuth2** authentication type, if required.

Example Configuration

The following is an example "**Authentication**" configuration for the **ConfluentOAuth2** type.

```
"Authentication" : {
   "Type" : "ConfluentOAuth2",
   "TenantId" : my-tenant-id",
   "ClientId" : "my-client-id",
   "ClientSecret" : "my-client-secret",
   "Scope" : "<https://<cluster-id>.<region>.confluent.cloud/>",
   "IdentityPool" : "pool-xxxxx",
   "TargetCluster" : "lsrc-yyyyyy" }
```

None

For the **None** authentication type, there are no additional values to specify.

Example Configuration

The following is an example "**Authentication**" configuration for the **None** type.

```
"Authentication" : {
   "Type" : "None"}
```

"Type":"None" specifies that no authentication will be used.

Note This field is valid for the Health Monitoring RestPublisher only, and not the **AzureSender** as this service always requires authentication. This type may be useful for sending REST health monitoring data to a generic endpoint that requires no authentication.

Environment Variable Placeholders

To avoid storing secret credentials directly in configuration files, environment variables are often used to inject these values into the application process. This technique is especially useful in Cloud environments.

Instead of specifying actual values in the configuration, you can use environment variable placeholders, which follow the following convention in which **EnvironmentVariable** is the name of the environment variable to use.

```
"${EnvironmentVariable}"
```

The following is a configuration example that uses environment variables.

```
"Authentication": {
   "Type": "${ENV-AUTH-TYPE}",
   "TenantId": "${ENV-TENANT-ID}",
   "ClientId": "${ENV-CLIENT-ID}",
   "ClientSecret": "${ENV-CLIENT-SECRET}" }
```

Usage Note

It is recommended that you use Azure Key Vault or another secure secret management solution to store sensitive information such as **ConnectionString** and **ClientSecret**, rather than including them directly in configuration files.

Performing a Bulk Extract

A *bulk extract* refers to a mass sending of events describing the current state of object instances in the database through the event stream without waiting for an update on these instances.

The most common reason for this is to fill a data sink that sits downstream in your event streaming pipeline.

Use the **updateObjectAndSlobOrBlobEditions** method of the **Object** class to perform a one-time bulk extract to push the current state of a large number of object instances through the event stream without waiting for an updating transaction for these instances.

Before performing a bulk extract, make sure that it is necessary for your use case and that you have considered some of the performance concerns listed later in this section.

Before you attempt any bulk extract operations, ensure that the **JadeChangeCaptureApp** application is running and configured correctly and that the classes you want to extract are specified in the Change Capture Specification file. For details, see "Change Capture Specification File", elsewhere in this document.

After configuring the Change Capture Specification file, write an appropriate script to perform the bulk extract and run it. The following is an example of a script to perform the bulk extract with comments providing additional context.

```
bulkExtractClass();
vars
    instance: Object;
    instanceArray: ObjectArray;
    maximumInstances: Integer64;
begin
    //Collect all the instances of this class
    create instanceArray transient;
```

```
    /*Set the maximum number of instances to extract. Be careful, Max_Integer
    (max value allowed for this method) is used for demonstration, but make
    sure you know how many you're about to extract before you start*/
    maximumInstances := Max_Integer;
    C1.allInstances(instanceArray, maximumInstances, false); /*Last parameter
    indicates we don't want instances that are subclasses*/
    beginTransaction;
    foreach instance in instanceArray do
        if app.isValidObject(instance) then /*Make sure the instance hasn't been
       deleted by a concurrent transaction*/
            instance.updateObjectAndSlobOrBlobEditions(instance);
            //Here, any object will do as a receiver. Only the parameter matters
      endif;
    endforeach;
    commitTransaction;
epilog
    delete instanceArray;
end;
```

Note the comments in the previous method example, specifically:

- Consider carefully the value specified as the maximum number of instances. Extracting a large number of instances will likely have a significant performance impact on both Jade and the event stream. Make sure you have an idea of the number of instances and the performance impact *before* you attempt a bulk extract on a production system. You could mitigate the performance impact by:
  - Splitting the transaction; for example, if 10,000 instances are too many to extract at once, you could get the loop to extract the first 1,000 instances in one transaction, the next 1,000 in another transaction, and so on.
  - Filtering the instances you want to extract; for example, call the **updateObjectAndSlobOrBlobEditions** method only on instances where a timestamp property of the instance falls within a certain range.
- The **allInstances** method returns all of the persistent instances that exist at the time that the call is made. On an active system, between this call and the **updateObjectAndSlobOrBlobEditions** method calls, new objects of the class could have been created or some of them could have been deleted.
  - Newly-created objects will not be captured by this script, but will be captured by the event stream as a **create** event so they will end up being extracted.
  - Objects could be deleted making references within the **instanceArray** invalid. The script in the previous method example checks the reference is valid to avoid an error 4 (*Object not found*).
  - Overall on an active system, due to performance impact and potential changing state, we recommend taking the system offline for a large bulk extract. It is possible to perform a bulk extract without first taking the system offline, but consider the potential impact carefully.
- The last parameter of the **allInstances** method:
  - Specifies whether to include instances that are subclasses of the specified class. If you want to include subclasses, make sure these classes are specified in the Change Capture Specification file (use the **recursiveSubClasses** option to include subclasses of a specified class).
  - Does not find subclasses recursively (unlike the **recursiveSubClasses** option). You must specify any classes that you want to extract farther down the hierarchy in your script.

  In addition, if you only want to extract some subclasses using this script, we recommend specifying them individually as there will be an unnecessary performance cost to calling the method on classes you are not extracting.

The **allInstances** method and the **recursiveSubClasses** option do not search for subclasses across schemas. Write your own scripts to find these subclasses if you want to include them in the bulk extract.

Applies to Version: 2025.0.01 and higher

Chapter 4 Installing and Running the Event Stream Producer

This chapter covers the following topics.

- Installing the Event Stream Producer
- Event Stream Producer Runtime Environments
- Running Event Stream Producer Applications
  - Running the Event Stream Producer (ESP) as an Application
  - Running the Event Schema Generator (ESG)
- Health Monitoring Data Specification

Note To run the Event Stream Producer, you need a Jade system that is running on a version of Jade 2022.0.04 (Service Pack 3) and higher.

Installing the Event Stream Producer

Extract the .**zip** contents to a suitable directory, but *not* to the main **Jade\bin** directory. (To get the Event Stream Producer (ESP) components, download the ZIP file from the Event Streaming tile on the **Extensions** page of the Jade World Developer Centre; that is, https://www.jadeplatform.com/developer-centre/extensions.)

The Event Stream Producer (ESP) needs read access to files created by the source Jade system.

For first-time use and initial testing, we recommend that you install the ESP on the same machine as the source Jade system.

Event Stream Producer Runtime Environments

The prerequisites for all Event Stream Producer (ESP) runtime scenarios are as follows.

- Windows operating system for the Jade Platform database and capture process
- Event Stream Producer (ESP) component installed
- Azure Command Line Interface (Azure CLI) for Azure scenarios
- One of the following Infrastructure as Code (IaC) tools
  - Terraform
  - Pulumi
  - Ansible
  - Azure Bicep

Tip Using IaC tools like Terraform, Pulumi, Ansible, or Azure Bicep allows for version-controlled, repeatable deployments. This is especially useful for maintaining consistency across development, testing, and production environments. Always follow security best practices when managing infrastructure and credentials.

Scenario 1: Development and Testing with Local Single Node Kafka Cluster

This scenario sets up a local development environment using a single-node Kafka cluster, preferably using Kraft instead of ZooKeeper.

The development and testing with local single node Kafka cluster steps are as follows.

1. Install WSL2 on Windows. Follow the official Microsoft guide.
2. Set up Kafka in WSL2.
3. Start Kafka server.
4. Configure ESP to connect to local Kafka.
5. Run ESP and test event streaming. (The ESP and verify events are produced using a consumer, or tool, that can consume the stream.)

Additional Resources

Additional resources are as follows.

- Kafka documentation
- WSL2 documentation

Scenario 2: Windows VM in Azure with Azure Event Hubs

This scenario sets up ESP in an Azure Windows Virtual Machine (VM), connecting to Azure Event Hubs with Schema Registry and Azure Blob Storage.

The Windows VM in Azure with Azure Event Hubs steps are as follows.

1. Create a Terraform or Azure Bicep script to provision:
    a. Windows VM in Azure
    b. Azure Event Hubs namespace and hub
    c. Azure Schema Registry
    d. Azure Blob Storage account
2. Run the IaC script to create the infrastructure.
3. Install and configure ESP on the Windows VM. Consider using Ansible for this step.
4. Configure ESP to connect to Event Hubs, Schema Registry, and Azure Blob Storage.
5. Run ESP and verify event streaming.

Additional Resources

Additional resources are as follows.

- Azure Event Hubs documentation
- Azure Blob Storage documentation
- Terraform Azure Provider
- Azure Bicep documentation

Scenario 3: Windows VM in Azure with Confluent Cloud

This scenario sets up ESP in an Azure Windows VM, connecting to Confluent Cloud with Confluent Schema Registry and Azure Blob Storage.

The Windows Virtual Machine (VM) in Azure with Confluent Cloud steps are as follows.

1. Create a Terraform script to provision:
   a. Windows VM in Azure
   b. Confluent Cloud resources (Kafka cluster, Schema Registry)
   c. Azure Blob Storage account
2. Run the Terraform script to create the infrastructure.
3. Install and configure ESP on the Windows VM. Consider using Ansible for this step.
4. Configure ESP to connect to Confluent Cloud, Schema Registry, and Azure Blob Storage.
5. Run ESP and verify event streaming.

Additional Resources

Additional resources are as follows.

- Confluent Cloud documentation
- Confluent Schema Registry documentation
- Azure Blob Storage documentation
- Terraform Confluent Provider
- Terraform Azure Provider

Running Event Stream Producer Applications

This section describes running the:

- Event Stream Producer (ESP) as an Application
  - Running the ESP as a Windows Service

- Event Schema Generator (ESG)

Running the Event Stream Producer (ESP) as an Application

The syntax for running the ESP as an application is as follows.

```
JadeEventProducer <config-file-name> [Producer Name]
```

The **JadeEventProducer** parameters are listed in the following table.

Parameter	Description
<config-file-name>	Path to the event stream producer configuration file. **Note** If you are running multiple copies of the ESP, you should use different configuration (**config.json**) files for each Producer, specifying different log files so that multiple producers do not write to the same log file.
Producer name	Optional name for this producer. The default producer name is **DEFAULT**. The producer name is: ■ Represented in the log file and is useful when distinguishing between messages from producers running on different databases. ■ Displayed as the title of the Console window in which it is running, which enables you to distinguish between different producers. If you try to run more than one producer reading from the same change capture directory with the same name, this will be detected, logged, and the producer shut down.

The following is an example of running the ESP as an application.

```
JadeEventProducer.exe config\producer.config.json PRODUCER1
```

Refer to "Event Stream Producer Configuration" in Chapter 3 for details about setting up the Producer configuration file.

For details about running the ESP as a Windows service, "Running the ESP as a Windows Service", later in this chapter.

Running the ESP as a Windows Service

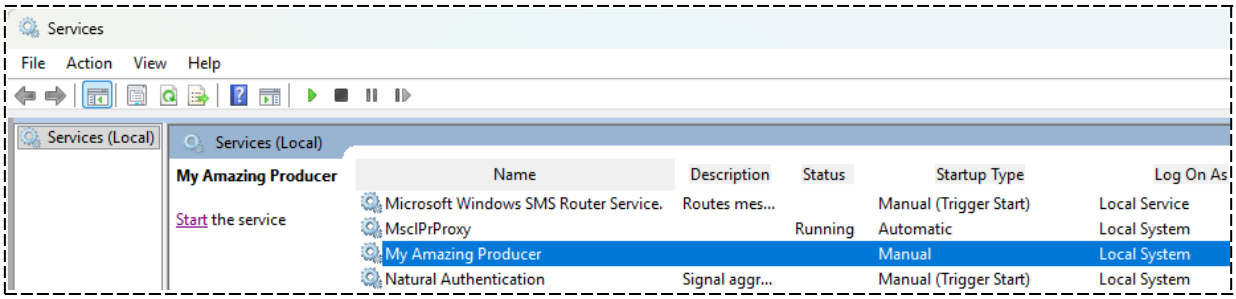
You can run the **JadeEventProducer.exe** as a Windows service. A Windows service is more suitable for long-running background tasks that require continuous execution. The service is managed by the Windows Service Control Manager (SCM) and can be started, stopped, paused, and restarted using tools such as the Services application or **sc.exe**. The Services application must be run with administrator rights.

Install the ESP as a service. In the command line, insert the option **--install** (or **-i**) before the **Producer name** parameter, as shown in the following example.

```
./JadeEventProducer.exe "My config file.jsonc" --install "My Amazing Producer"
Service installed successfully
```

For the installation to succeed, you must run the command with administrator rights.

After the ESP is installed as a Windows service, it is displayed in the Services application with the same name as the one specified in the **Producer name** parameter, as shown in the following image.



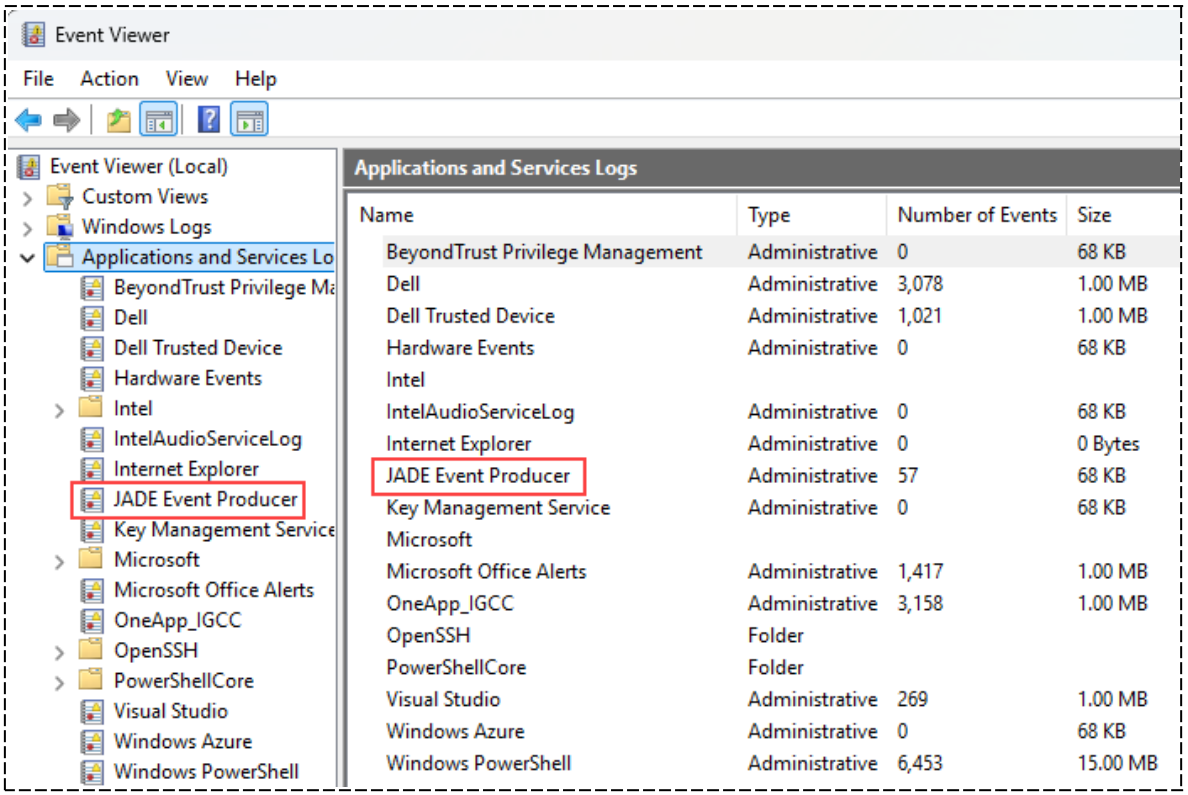
The Producer can be installed multiple times with different names and configurations for different systems.

By default, the ESP service is installed as a **Manual** start-up type in a **Stopped** state. It has a command line with the same parameters as it had when installed; that is, the configuration file when installed always applies to this service.

The Producer can be started and stopped with this Services application or variations of the **sc** command; for example, the service can be removed with the following command (when run as administrator).

```
sc delete "My Amazing Producer"
```

There is no command line output for a Windows service. If there are issues regarding the service itself, error messages can be written to the application event log under **JADE Event Producer**, which you can view with the Windows Event Viewer, as shown in the following image.



The destination for log messages (including errors) is defined in the **Logger** or **Loggers** section of the configuration file.

When the Event Producer starts, logging may not yet have begun, in which case error messages are sent to **stderr**. When the Event Producer is running as a service, these logging messages are not usually visible. To see messages sent to **stderr**, redirect the output messages to a file; for example, when installing the service, add a windows redirect command such as the following on the command line.

```
2>c:\logging\startup_errors.log
```

In all other ways, the Event Producer behaves in the same ways as it does when it is run from the command line.

Running the Event Schema Generator (ESG)

The command to run the Event Schema Generator (ESG) is as follows.

```
JadeEventSchemaGenerator <input-file-name>
                         <output-root-folder>
                         [<schema-name>]
                         [<class-name>]
```

The ESG parameters are listed in the following table.

Parameter	Specified
<input-file-name>	Path to the Jade Audit Access description file that will be used as input
<output-root-folder>	Directory where the generated JSON schema files will be saved
<schema-name>	Optional name of the schema that contains the class that is to be processed
<class-name>	Optional name of the class for which to generate a specification

Example of Generating a Schema File for One Class

```
JadeEventSchemaGenerator description20240725135130.txt ./event-schemas MySchema
MyClass
```

In this example, the **JadeEventSchemaGenerator** reads the **description20240725135130.txt** file, generates a single JSON schema file for the specified class, and then saves it in **./event-schemas/MySchema/**. The generated file is named **MyClass_v0001.schema.json** (for version 0001 of the class).

Example of Generating a Schema File for All Classes

```
JadeEventSchemaGenerator description20240725135130.txt ./event-schemas
```

In this example, the **JadeEventSchemaGenerator** reads the **description20240725135130.txt** file, generates a JSON schema file for each user class, and then saves them in the **./event-schemas/** folder structure specified in "Output Files" in the following section.

Note All collection classes and system classes are ignored.

Output Files

The output files are organized under the following folder structure.

```
<output-root-folder>
   <schema-name>
```

<schema-name>

Each JSON schema file is stored in the subfolder for its respective Jade schema and has the following file name.

*<class-name>*_v*<class-version>*.schema.json

Health Monitoring Data Specification

The Health Monitor, if enabled, sends regular health status information for the producer component of Jade Event Streaming to a specified URL or to a broker management topic.

The consumer of the health status information can monitor the general health status of this component. If configured, the information is sent at regular intervals for the whole lifetime of the producer.

The "**HealthMonitor**" section provides configuration settings that enable you to run the Health Monitor for a producer.

The health monitoring data are sent as a JSON string described by the following schema.

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties":
    {
        "Producer":
        {
            "type": "string"
        },
        "TimeGenerated":
        {
            "type": "string",
            "format": "date-time"
        },
        "ProducerStarted":
        {
            "type": "string",
            "format": "date-time"
        },
        "LastEventSent":
        {
            "type": "string",
            "format": "date-time"
        },
        "EventsSentToDeadLetter":
        {
            "type": "integer"
        },
        "ErrorsInPeriod":
        {
            "type": "integer"
        },
        "EventsInPeriod":
        {
            "type": "integer"
        },
        "TotalSenders":
        {
            "type": "integer"
```

```
            },
            "AliveSenders":
            {
               "type": "integer"
            }
        },
        "required":
        [
            "Producer",
            "TimeGenerated"
        ]
    }
```

The properties in the health monitor data message are described in the following table.

Property	Description
Producer	Name given to a producer in the command line for that producer. If no value is specified, it has the value "**DEFAULT**".
TimeGenerated	Time the data message was created. This is a string in ISO-8601 format. The time is fully specified as local time (to milliseconds) with time zone offset; for example, "**2025-06-19T14:03:28.194+12:00**" for NZST. **Notes** In versions of Windows before Windows 10 19H1 or Windows Server 2022: ■ Time zone information is not available. In these Windows versions, the timestamp format looks like "**2025-06-19T14:03:28.194**" (which is the local time of the Event Producer). ■ Azure Monitor cannot be used as a valid endpoint. This is because published timestamps are not generated in ISO-8601 format, which prevents health records from being ingested by Azure Monitor.
ProducerStarted	Time the producer was last started. (Uses the same time format as **TimeGenerated**.)
LastEventSent	Time of the last health monitoring message for a period that contained any events.
EventsSentToDeadLetter	Number of events that have been sent to the **FileReader** section **UndeliverableDirectory** in the lifetime of this producer run. (It is not necessarily the same as the number of events in that directory.)
ErrorsInPeriod	Count of any errors that have occurred while sending events to the broker since the last health monitoring message. An error occurring does not necessarily indicate that an event was not eventually successfully sent; a send might be retried successfully.
EventsInPeriod	Count of event sends to all topics since the last health monitoring message. A single event sent to two topics would increment the count by two.
TotalSenders	This value is always the same as the value specified for the **NumberOfSenders** parameter in the **AzureSender** section. It is published here so that it can be read by monitoring software automatically. **Note** This field applies to and is published only for an AzureSender.

Property	Description
AliveSenders	Count of senders that are still responsive. This value is usually the same as the **TotalSenders** value. If a sender stops responding: ■ The **AliveSenders** count reduces. ■ An info-level message is logged. If the **AliveSenders** count equals zero (**0**), there are no senders working and the Producer must be restarted. **Note** This field applies to and is published only for an AzureSender.

Applies to Version: 2025.0.01 and higher

Chapter 5 Event Structure and Schemas

This chapter covers the following topics.

- Overview
- Event Structure and Definition
- Event Schema
- JSON Schema Generation

Overview

The Event structure outlines the format of each event message, detailing the fields that are expected. It consists of two key sections, as follows.

- A common header, containing metadata and control information.
- A body (or 'object') section, which houses the actual data for the event, typically mapped one-to-one with each Jade property.

The Schema definition specifies the exact type, size, and format of each field for each class of event message produced by the event producer. This ensures that consumers can accurately deserialize and interpret the message. The Schema definition covers both the common header fields and the class-specific body fields within the 'object' section.

Event Structure and Definition

The following shows an example of the event that creates the **Account** class in the **BankingModelSchema**.


```
{
   "eventId" : "21fe3632-ff91-ef11-9f3c-005056a9d6b5.8445.1.1",
   "schemaId" : "8445.v1",
   "eventType" : "JadeSoftware.ObjectStorage.ObjectCreated",
   "eventTime" : "2024-10-24T11:58:49.872Z",
   "eventTimezoneOffset" : "+13:00",
   "eventSequenceNumber" : 5736406,
   "contentType" : "application/json",
   "transactionId" : 570,
   "objectType" : "BankModelSchema::Account",
   "object" : {
      "Jade.oid" : "8445.1",
      "Jade.instanceId" : 1,
      "Jade.edition" : 1,
      "active" : false,
      "date" : "2013-01-01",
      "district_id" : 55,
      "frequency" : "Monthly Issuance",
      "id" : "A00000576",
      "myBank" : {
         "Jade.objectType" : "BankModelSchema::Bank",
```

```
            "Jade.instanceId" : 1,
            "Jade.oid" : "8443.1"
         },
         "myDistrict" : {
            "Jade.objectType" : "BankModelSchema::District",
            "Jade.instanceId" : 55,
            "Jade.oid" : "8448.55"
         }
      }
   }
```

Event Properties

The following table lists the required event properties.

Property	Type	Description
eventId	String	Unique identifier for the event. If a duplicate event is re-sent (for example, due to a network error), it could have the same identifier. Consumers may assume that events with an identical source and **eventId** property are duplicates. The **eventId** property format is **<*database-unique-ID*>-<*object-OID*>-<*object-edition*>**; for example, **"560df1f8-3556-ef11-8a20-005056a99710.2051.4501.1"**.
schemaId	String	The schema identifier associated with the event. The **schemaId** value is a string that has the following formats. ■ **Azure**: a GUID string ■ **Kafka**(Confluent): An integer starting at **1**, stored as a string ■ **None**: <*class-number*>.v<*class-version-number*>
eventType	String	Type of event that occurred (that is, **ObjectCreated**, **ObjectUpdated**, **ObjectDeleted**, or **TransactionCommitted**); for example, **"JadeSoftware.ObjectStorage.ObjectCreated"**.
eventTime	String	Timestamp of when the state change was audited in the database journal; for example, **"2024-08-12T02:11:02.873Z"**.
eventTimezoneOffset	String	Timezone offset for the event time; for example, **"+12:00"**.
eventSequenceNumber	Integer	Monotonically increasing sequence number that can be for ordering events. These are unique for the source database identified by the **<*database-unique-ID*>** part of the **eventId** property; for example, **5512494**.
contentType	String	Content type of the event payload. The value is always **"application/json"** for events serialised to a JSON format; that is, **"application/json"**.
transactionId	Integer	The database unique transaction identifier associated with the event. This is the identifier of the database transaction that performed the create, update, or delete operation. An example of the **transactionId** property is **756**.

Property	Type	Description
objectType	String	Type of object contained in the embedded object property (for example, **BankModelSchema::Account**, **BankModelSchema::District**). The **objectType** property format is **<*schema-name*>::<*class-name*>**; for example, **"BankModelSchema::Account"**.
object	Object	Embedded object representing the event payload.

Object Properties

This section covers the properties of the **object** property.

Jade Object Properties

The following table lists the required Jade properties of the **object** property.

Property	Type	Description
Jade.edition	Integer	Update edition for the object. Starts off as **1** when an object is created and is incremented each time it is updated, once for each transaction; for example, **1**.
Jade.instanceId	Integer	Instance identifier for the object; for example, **4501**.
Jade.oid	String	Unique identifier for the object within the database; for example, **"2051.4501"**.

User-Defined Object Properties

The Jade properties are followed by the static properties defined in the user class. For reference, the following table lists examples of user-defined **object** property descriptions.

Property	Type	Description
active	Boolean	Indicates whether the object is currently active.
date	String	Date associated with the object.
district_id	Integer	Identifier for the district associated with the object.
frequency	String	Frequency of the object issuance.
id	String	Unique identifier for the object.
incarnation	Integer	Incarnation identifier for the object.
myBank	ObjectReference	Reference to the bank associated with the object.
myDistrict	ObjectReference	Reference to the district associated with the object.

Object Reference Properties

The following table lists the required object reference properties.

Property	Type	Description
Jade.instanceId	Integer	Instance identifier for the object; for example, **132**.

Property	Type	Description
Jade.objectType	Integer	Type of object being referenced. The **Jade.objectType** property format is <***schema-name***>::<***class-name***>; for example, **"BankModelSchema::Account"**.
Jade.oid	String	Unique identifier for the object within the database; for example, **"2054.132"**.

Event Schema

The Jade Platform Event Stream Producer (ESP) uses a JSON schema to describe database state change events. For details about the JSON schema specification, see https://json-schema.org/specification.

The JSON schema defines the structure for describing database change events in a Jade Platform database. It serves as a template for generating valid JSON payloads that represent these events.

The schema is composed of the following key components.

1. Root-level properties
2. Definitions section, which contains nested schemas

Root-Level Properties

The following table lists the root-level properties in the JSON schema.

Property	Description
$schema	Specifies the JSON schema draft version; for example, **"http://json-schema.org/draft-07/schema#"**
title	Provides a human-readable title for the schema
type	Defines the root-level object type
properties	Lists all of the properties of the root object
required	Specifies which properties are mandatory

The following table lists details of the **properties** property.

properties Detail	Description
eventId	String type
schemaId	String type
eventType	String type
eventTime	String type with date-time format
eventTimezoneOffset	String type
eventSequenceNumber	Integer type
contentType	String type
transactionId	Integer type
objectType	String type
object	Object type referencing another schema

Definitions Section

The **"definitions"** section of the JSON schema contains nested schemas for complex types.

Object Definition

The **"Object"** defines the structure of the main event payload object.

ObjectReference Definition

The **"ObjectReference"** defines the structure for referencing other objects.

Constraints

The following constraints apply to the JSON schema.

- Uses JSON Schema draft-07 standard
- Implements recursive references using **$ref**
- Applies type definitions and constraints to each property

The following is an example of a JSON schema.

```
{
    "$schema" : "<http://json-schema.org/draft-07/schema#>",
    "title" : "BankModelSchema.Account.v1",
    "type" : "object",
    "properties" : {
    // ... properties definitions ...
    },
    "required" : ["eventId", "schemaId", "eventType", "eventTime",
          "eventTimezoneOffset", "eventSequenceNumber", "contentType",
          "transactionId", "objectType", "object"],
    "definitions" : {
        "Object" : {
        // ... Object definition ...
    },
    "ObjectReference" : {
        // ... ObjectReference definition ...
       }
    }
}
```

JSON Schema Generation

The automatic generation of Jade Audit Access (JAA) description files must be enabled by setting the **UseJournalDescriptions** parameter to **true** in the [PersistentDb] section of the Jade initialization file. This parameter is **false** by default. For details, see "**UseJournalDescriptions**", in the *Jade Initialization File Reference*.

The **JadeEventSchemaGenerator** is a useful utility for previewing the Event Schema file that corresponds to a specific **Event** class. The **JadeEventProducer** itself automatically generates and registers these Event Schema files on demand with the **SchemaRegistry**, if configured to do so.

To support this functionality, the "**Catalog**" module uses Jade Audit Access (JAA) description files to discover the classes and properties recorded in the journal files.

The **JadeEventSchemaGenerator** is designed to create JSON schema files suitable for Kafka Event Streaming or Azure Event Hubs. It takes as input a Jade Audit Access description file and produces the corresponding JSON Event Schema files describing the structure of user data in the journal files.

The Jade Audit Access description file is automatically created at the completion of a reorganization, following the change in structure of persistent classes. (For details about running the ESG, see "Running the Event Schema Generator (ESG)" under "Running Event Stream Producer Applications", in Chapter 4.

JSON schema files for Event Streaming are used to define the structure, content, and semantics of messages (events) that are exchanged within an Event Streaming-based system. These JSON schemas ensure that the data produced by one application and consumed by another adheres to a specific format, enabling seamless data exchange and validation across distributed systems.

Chapter 6 External Storage Providers

This chapter covers the following topics.

- Overview
- External Storage Provider Architecture
- External Storage Providers
  - File System Provider
  - Azure Blob Storage Provider

Overview

In the context of Event Streaming, an external storage provider is a third-party service or system used to store, manage, and sometimes process event data outside of the primary Event Streaming platform. This provider typically offers scalable storage solutions, allowing event data to be retained for longer periods or shared across different applications and services.

External storage providers are often integrated with Event Streaming platforms to:

- Persist large volumes of data for historical analysis or auditing.
- Enable downstream processing by other applications that need access to the event data.
- Enhance data durability by ensuring data is safely stored beyond the real-time processing scope.

See also the "FileStorageProvider" section under "Event Stream Producer Configuration", in Chapter 3.

External Storage Provider Architecture

Event streaming platforms such as Kafka and Azure Event Hubs are primarily designed for real-time data processing and have inherent limitations on message size. By default, these platforms impose message size limits of approximately 1MB. This limitation poses a significant challenge when dealing with scenarios where state changes involve large blob or slob data. Examples of large data are documents and various binary data content such as multimedia data.

These large data items cannot be efficiently represented within the constraints of the default message size limit. An External Storage Provider is a potential solution that allows storing large data externally while still maintaining a connection to the parent event.

For details about configuring your size thresholds by setting limits for when large binary (blob) and string (slob) objects should be stored externally instead of being included directly in the JSON output, see the "SizeThresholds" section under "Event Stream Producer Configuration", in Chapter 3.

Key Features

The external storage provider has the following key features.

- Enables flexible storage options (for example, Azure Blob or the local file system)
- Provides a standardized interface for integrating custom storage solutions
- Maintains JSON schema compliance for event representation
- Enhances scalability by offloading large data to external storage
- Uses a special prefix (**$**) for property names to avoid conflicts

Benefits

The external storage provider has the following benefits.

- Supports representing database state changes that exceed Kafka or Event Hubs message size limits
- Increased modularity and extensibility
- Clear association of URIs with their corresponding object properties

External Storage Providers

External Storage Providers in the Event Stream Producer architecture serve as bridges between the core system and various external storage solutions. They implement the **IExternalStorageProvider** interface, providing methods for storing binary and text data associated with blob or slob attributes. These providers enable the Event Stream Producer to offload large data items to external storage systems, overcoming message size limitations in event streaming platforms like Kafka and Azure Event Hubs.

Supported storage types include local file systems and Azure Blob Storage.

All providers implement the **IExternalStorageProvider** interface.

File System Provider

The File System Provider (the "**FileStorageProvider**" section) requires the **RootDirectory** parameter, which specifies the root directory for storing files.

The File System Provider:

- Uses a structured directory hierarchy based on event timestamp and transaction identifier.
- File naming convention includes event identifier and property name.
- Implements error handling and logging.
- Error handling: The implementation includes error handling and logging mechanisms.
- Data management: The hierarchical directory structure (year-month/transaction identifier) is designed to help with data aging scenarios, including archiving and bulk deleting historical data.

Directory Structure

The hierarchical directory structure (year-month/transaction identifier), shown in the following example, is designed as an aid to archiving and aging historical data.

```
RootDirectory\
├── YYYY-MM\
│   ├── TransactionID1\
│   │   ├── [EventID1]__PropertyName1.blob
│   │   └── [EventID1]__PropertyName2.slob
│   └── TransactionID2\
│       ├── [EventID2]__PropertyName1.blob
│       └── [EventID2]__PropertyName2.slob
└── YYYY-MM\
└── ...
```

For more details, see "FileStorageProvider" under "Event Stream Producer Configuration", in Chapter 3.

Azure Blob Storage Provider

The Azure Blob Storage Provider provides configuration for the Azure Blob Storage provider, used for storing over-threshold size blob and slob data in Azure Blob Storage. (For Azure-related configurations, use the Azure Portal or Azure Command Line Interface (CLI).)

The parameters of the "**AzureBlobStorageProvider**" section are listed in the following table.

Parameter	Required?	Description
connectionString	Yes	Azure Storage account connection string.
containerName	Yes	Name of the Azure Blob container.
stagingDirectory	Yes	Staging directory where blob and slob files are stored before they are uploaded to Azure Blob Storage. A directory hierarchy similar to that used by the **FileStorageProvider** is created under the specified **StagingDirectory**. Staged capture files are removed after they have been uploaded to blob storage. **Notes** This unique directory name cannot be shared if multiple **JadeEventProducer.exe** applications are run. As the temporary directories created to house the blob files are not cleaned up by the application, you need to maintain them yourself at this stage.
blobPrefix	No	Optional prefix added to all blob directory names; for example, **"events"**.

For more details, see "AzureBlobStorageProvider" under "Event Stream Producer Configuration", in Chapter 3.

Chapter 7 Schema Registry and Caching

This chapter covers the following topics

- Schema Caching
- Schema Registry
- Azure Schema Registry

Schema Caching

The Schema Caching component provides the following main benefits.

- Efficiently looks up the Event Schema and its associated **"schemaId"** to stamp each event.
- Persists Event Schema definitions to a directory structure, enabling alternative processing outside of the Schema Registry mechanism.
- Manages the generation and registration of Event Schema definitions for events produced by the Event Producer, ensuring that schemas are registered before events are sent to EventHub or Kafka.

Schema Registry

The Schema Registry provides a centralized repository for managing and validating schemas used for topic message data, as well as for the serialization and deserialization of data over the network. Producers and consumers use schemas to ensure data consistency and compatibility as the schemas evolve.

The Schema Registry is a key component of data governance, helping ensure data quality, adherence to standards, visibility into data lineage, auditing capabilities, cross-team collaboration, efficient application development, and system performance.

As event stream data evolves, schemas change, but the Schema Registry manages these schema versions to ensure compatibility between producers and consumers. It guarantees that the correct schema definitions are available as schemas evolve.

Azure Schema Registry

The Azure Schema Registry is a variant of Schema Registry designed to work with Azure EventHub. It supports schemas in JSON format. (For details, see Chapter 5, Event Structure and Schemas.)

Azure Schema Registry assigns a GUID-based **"schemaId"** to each unique schema definition it stores. Multiple versions of a schema can be registered, and if a new version differs from the previous one, the version number is incremented. These versions can be viewed in the Azure portal as they are registered.

Note Some service tiers have constrained limits on the total number of schemas allowed.

Appendix A Glossary

The terms used in this document are listed in the following table.

Term	Definition
at-least-once delivery	In case of failures that lead to message loss or taking too long to recover from a failure, messages are retransmitted to assure they are delivered at least once.
benchmarking	Process of testing and evaluating the performance of a system or component.
change capture	Process of identifying and capturing changes made to a data source.
Cloud events	Specification for describing event data in a common way.
delta database	Database that stores and manages changes (deltas) to data over time.
event	Record of a state change in the system.
event hubs	Multi-protocol event streaming engine provided by Microsoft as part of its Azure Cloud Service.
event schema	Structure and format definition for events in a streaming system.
Event Schema Generator (ESG)	Utility for generating Event Schema files for a Jade **Event** class.
Event Stream Producer (ESP)	Component that generates and sends events into a streaming system.
event streaming	Practice of capturing, processing, and storing real-time data streams.
event structure	Defined format and composition of an event in a streaming system.
external storage provider	System or service that provides storage capabilities outside the main event streaming infrastructure.
Jade event	Specific type of event structure used in the system described in the document.
JSON (JavaScript Object Notation)	Lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. Used for serializing objects before transmission over the web.
JSON schema	Defines the structure and validation rules for JSON data.
Kafka	Distributed event streaming platform and API provided and defined by the Apache Software Foundation.
RSA (Rivest–Shamir–Adleman encryption algorithm)	Validates the signature of a JSON Web Token.
schema caching	Practice of temporarily storing schemas in memory for quick access and improved performance.
schema registry	System for storing and retrieving schemas, often used in event streaming architectures.
serialization	Process of converting data structures or objects into a format that can be easily stored or transmitted.

Term	Definition
serialization error handling	Process of managing and responding to errors that occur during the serialization of data.
topic	Category or feed name to which records are published in a streaming system.
URI (Uniform Resource Identifier)	Generic term for all types of names and addresses that refer to objects (for example, a URL) on the Internet. Commonly used term when talking about REST.