DigitalEdge™

SDK Guide

Version 1.2.1

July 2014

leidos

DISCLAIMER OF WARRANTY AND LIMITATION OF LIABILITY

The Software accompanying this Documentation is provided with the Limited Warranty contained in the License Agreement for that Software. Leidos, its affiliates and suppliers, disclaim all warranties that the Software will perform as expected or desired on any machine or in any environment. Leidos, its affiliates and suppliers, further disclaim any warranties that this Documentation is complete, accurate, or error-free. Both the Software and the Documentation are subject to updates or changes at Leidos' sole discretion. LEIDOS, ITS LICENSORS AND SUPPLIERS MAKE NO OTHER WARRANTIES, WRITTEN OR ORAL, EXPRESS OR IMPLIED RELATING TO THE PRODUCTS, SOFTWARE, AND DOCUMENTATION. LEIDOS, ITS LICENSORS AND SUPPLIERS DISCLAIM ALL IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, USE, TITLE, AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS. In no event shall Leidos, its affiliates or suppliers, be liable to the End User for any consequential, incidental, indirect, exemplary, punitive, or special damages (including lost profits, lost data, or cost of substitute goods or services) related to or arising out of the use of this Software and Documentation however caused and whether such damages are based in tort (including negligence), contract, or otherwise, and regardless of whether Leidos, its affiliates or suppliers, has been advised of the possibility of such damages in advance. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAWS, END USER ACKNOWLEDGES AND AGREES THAT LEIDOS AND ITS AFFILIATES AND SUPPLIERS IN NO EVENT SHALL BE RESPONSIBLE OR LIABLE TO THE END USER FOR ANY AMOUNTS IN EXCESS OF THE FEES PAID BY THE END USER TO LEIDOS. LEIDOS SHALL NOT BE RESPONSIBLE FOR ANY MATTER BEYOND ITS REASONABLE CONTROL.

LEIDOS PROPRIETARY INFORMATION

This document contains Leidos Proprietary Information. It may be used by recipient only for the purpose for which it was transmitted and will be returned or destroyed upon request or when no longer needed by recipient. It may not be copied or communicated without the advance written consent of Leidos. This document contains trade secrets and commercial or financial information which are privileged and confidential and exempt from disclosure under the Freedom of Information Act, 5 U.S.C. § 552.

TRADEMARKS AND ACKNOWLEDGMENTS

Private installations of DigitalEdge are powered by Eucalyptus®.

Public cloud installations of DigitalEdge are powered by Amazon Web Services™.

The following list includes all trademarks that are referenced throughout the DigitalEdge documentation suite.

Adobe, Flash, PDF, and Shockwave are either registered trademarks or trademarks of Adobe Systems Incorporate in the United States and/or other countries.

Amazon Web Services, AWS, Amazon Elastic Compute Cloud, Amazon EC2, EC2, Amazon Simple Storage Service, Amazon S3, Amazon VPC, Amazon DynamoDB, Amazon Route 53, the "Powered by Amazon Web Services" logo, are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.

Apache, Archiva, Cassandra, Hadoop, Hive, HBase, Hue, Lucene, Maven, Apache Phoenix, Solr, Zoie, ActiveMQ are all trademarks of The Apache Software Foundation.

ArcSight is a registered trademark of ArcSight, Inc.

CAS is copyright 2007, JA-SIG, Inc.

CentOS is a trademark of the CentOS Project.

Cloudera is a registered trademark of Cloudera, Inc.

CloudShield is a registered trademark of CloudShield Technologies, Inc. in the U.S. and/or other countries.

CTools are open-source tools produced and managed by Webdetails Consulting Company in Portugal.

Drupal is a registered trademark of Dries Buytaert.

Elasticsearch is a trademark of Elasticsearch BV, registered in the U.S. and in other countries.

Eucalyptus and Walrus are registered trademarks of Eucalyptus Systems, Inc.

Firefox is a registered trademark of the Mozilla Foundation.

The Groovy programming language is sustained and led by SpringSource and the Groovy Community.

H2 is available under a modified version of the Mozilla Public License and under the unmodified Eclipse Public License.

Hybridfox is developed and maintained by CSS Corp R&D Labs.

JUnit is available under the terms of the Common Public License v 1.0.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft, Windows, and Word are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

MongoDB and Mongo are registered trademarks of 10gen, Inc.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Pentaho is a registered trademark of Pentaho, Inc.

PostgreSQL is a trademark of The PostgreSQL Global Development Group, in the US and other countries.

PuTTY is copyright 1997-2012 Simon Tatham.

Sonatype Nexus is a trademark of Sonatype, Inc.

Tableau Software and Tableau are registered trademarks of Tableau Software, Inc.

Twitter is a registered trademark of Twitter, Inc.

All other trademarks are the property of their respective owners.

CONTACT INFORMATION

Leidos Franklin Center
6841 Benjamin Franklin Drive
Columbia, Maryland 21046

Email: DigitalEdgeSupport@Leidos.com

DigitalEdge Technical Support: 443-367-7770

DigitalEdge Sales Support: 443-367-7800

To submit ideas or feedback: https://www9.v1ideas.com/digitaledge/welcome

Contents

Introduction

The DigitalEdge SDK provides several APIs to easily create custom components for DigitalEdge's processing pipeline. The ingest pipeline consists of:

- Transports: Transports are protocols that are used to get data into DigitalEdge.
- Parsers: Parsers are the components which receive raw data and parse it into JSON records. Records in this form are also referred to as canonical format.
- Enrichment Processors: Processors enrich the JSON records with data from other databases and with algorithmic enrichments, producing multi-dimensional data that has been flattened into a single JSON record (also referred to as the enriched canonical format).
- Data sinks: Data sinks receive these enriched JSON records for post-processing, storing, or alerting.

Product documentation

DigitalEdge is a complex big data platform. The system comes with a complete set of documentation in PDF and HTML5 formats to help you master DigitalEdge:

Document	Use	Audience
Overview Guide	Basic information about the DigitalEdge platform, including architecture, concepts, and terminology; a must-read before working with any aspect of DigitalEdge	Anyone working with DigitalEdge in any capacity
Configuration Guide	Instructions for defining data models and building processing pipelines	Data Specialists, DigitalEdge Administrators
Operations Guide	Daily administration information, covering monitoring, managing, and modifying the platform	DigitalEdge Administrators
DigitalEdge SDK Guide	Reference for building custom plug-in components	Developers
Alerts API Guide	Reference for specifying data triggers and notifications for an alerting capability	Developers
Search API Guide	Reference for providing search services on a Lucene data sink node	Developers

Typographical conventions

The following style conventions are used throughout this documentation:

Type of Information	Style in Documentation
Code, commands, filenames	`code`
Cross references	Click to see this topic
Emphasis	*important point*
Hyperlinks	Click to go to this site
Notes, warnings, tips	
References to other documents	*Document Title*
Sample code blocks	`code`
Troubleshooting issue or problem	
Troubleshooting solution	
User input	*Italics*
User interface labels and controls	**Bold**
Variables	<*change-this-name*>

What the SDK includes

The DigitalEdge SDK includes the following components:

- Java library required for the APIs
- Javadoc for the APIs
- Sample components:
  - Transport code: `SimpleLocalFileTransportService.java`
  - Parser code: `SimpleLogParser.java`
  - Enrichment code: `SimpleAreaCalculationEnrichment.java`
  - Data sink code:`JsonFileDataSink.java`
  - Test driver code: `JsonFileDataSinkTest.java`
  - Test input file: `test.json`

Sample Code

To help you use and understand the SDK, several examples of custom-built components are provided here, with embedded comments. You can build:

- Transports
  - See "Creating a transport" on page 5
  - See "SimpleLocalFileTransportService.java" on page 6
- Parsers
  - See "Creating a parser" on page 9
  - See "SimpleLogParser.java" on page 10
- Enrichments
  - See "Creating an enrichment" on page 12
  - See "SimpleAreaCalculationEnrichment.java" on page 13
- Data sinks
  - See "Creating a data sink" on page 15
  - See "JsonFileDataSink.java" on page 17
  - See "JsonFileDataSinkTest.java" on page 21

Creating a transport

Transports are protocols used for getting data into the DigitalEdge system. DigitalEdge comes with many built in user-configurable transport options including an S3 File Transport, TCP, UPD, and URL. When a built in transport does not suffice, the DigitalEdge SDK helps you create a transport for specific needs. Once written, custom transport classes are integrated with DigitalEdge and made available through the DigitalEdge setup tools (e.g., **System Builder**).

To create a transport, you must write a class that implements the `com.saic.rtws.transport.TransportService` and `com.saic.rtws.common.util.Initializable` interfaces (`TransportService` extends `Initializable`, see the *Transport SDK Javadocs* for details). The key methods to implement are:

TransportService

`execute()`: Retrieve data from source, pre-process (if needed), and forward data into DigitalEdge. This method should continuously loop until terminate is called.

`terminate()`: Notify the `execute()` method to exit processing.

Implement the following:

Initializable

`initialize()`: Called before `execute()`. Use to create the connection to your datasource, initialize resources, etc.

`dispose()`: Called after `terminate()`. Use to clean up resources created in `initialize()`.

Your transport class should extend the `AbstractTransportService` class. This class provides implementations of the `initialize()` and `dispose()` methods and convenience methods for sending JMS messages into the DigitalEdge system.

If your transport requires user configurable parameters, use the `UserConfigured` annotation (`com.saic.rtws.core.framework.UserConfigured`) to annotate the setter methods of your transport's class properties. The DigitalEdge **System Builder** tool will read these annotations and display the annotated properties as user configurable fields in the tool.

SimpleLocalFileTransportService.java

```
package <your package>;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStreamReader;

import com.saic.rtws.core.framework.Description;
import com.saic.rtws.core.framework.UserConfigured;
import com.saic.rtws.transport.AbstractTransportService;
import com.saic.rtws.transport.TransportService;

@Description("Simple example of a DigitalEdge TransportService that reads files
       from a directory, reads the contents of the files," +
       " and pushes JMS messages into DigitalEdge")
public class SimpleLocalFileTransportService extends AbstractTransportService {

       private String watchedDirectory;
       private DirWatcherRunner runner;
       private int messagesSent = 0;

       public int getMessagesSent() {
              return messagesSent;
       }

       @UserConfigured(value = "/usr/local/data", description = "The local directory to
              poll for data to " + "transmit into the system.", flexValidator = {
              "StringValidator minLength=3 maxLength=63" })
       public void setWatchedDirectory(String val) {
              watchedDirectory = val;
       }

       public String getWatchedDirectory() {
              return watchedDirectory;
       }
```

```
	/**
	* Start this Transport service.
	* @see TransportService#execute()
	*/
	@Override
	public void execute() {
		runner = new DirWatcherRunner(watchedDirectory);
		Thread fileWatcher = new Thread(runner);
		try {
			fileWatcher.start();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	/**
	* Close any open resources and stop the transport.
	* @see TransportService#terminate()
	*/
	@Override
	public void terminate() {
		try {
			runner.setStop(true);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	/**
	*  Runnable to read the file stream, split into lines, and send as a JMS message.
	*/
	private class DirWatcherRunner implements Runnable {

		private String watchedDirectory;
		private boolean stop = false;

		public void setStop(boolean stop) {
			this.stop = stop;
		}
```

```
                public DirWatcherRunner(String watchedDirectory) {
                        this.watchedDirectory = watchedDirectory;
                }

                public void run() {
                        try {
                                while (!stop) {
                                        File watchDir = new File(watchedDirectory);
                                        if (watchDir.isDirectory()) {
                                                for (File f: watchDir.listFiles()) {
                                                        if (f.isFile()) {
                                                                FileInputStream stream = new FileInputStream(f);
                                                                BufferedReader reader = new BufferedReader(
                                                                        new InputStreamReader(stream, "UTF-8"));
                                                                String record = reader.readLine();
                                                                while (record != null) {
                                                                        // SendJMSMessage is the key method to push JMS
                                                                                messages into the system
                                                                        SendJMSMessage(record);
                                                                        messagesSent++;
                                                                        record = reader.readLine();
                                                                }
                                                                stream.close();
                                                                //f.delete();
                                                        }
                                                }
                                        }
                                }
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                }
        }

}
```

Creating a parser

Parsers extract and translate data sources into key-value pairs that are mapped to your input model. DigitalEdge comes with many built in user-configurable parsers that work with data formats like CSV, CEF, email, EXIF, and JSON. If you have a data source format that is not covered by the DigitalEdge parsers, you can use the SDK to write your own specific parser that can be plugged into the DigitalEdge framework and made available through the DigitalEdgesetup tools.

To create a parser, you must write a class that implements the `com.saic.rtws.core.framework.parser.Parser` interface, described below.

Parser

`clone()`: Create and return a copy of your parser.

`parse()`: Parse the incoming data source into a JSONObject. This method should contain the custom parsing logic required by your data source format.

`parseHeaders()`: Called before `parse()` by the DigitalEdge framework. Parse any headers in the incoming data source (headers are optional).

`setInputStream(java.io.InputStream)`: Called before `parse()` by the DigitalEdge framework. The InputStream associated with the incoming DataSource is passed as an argument.

`setStreamProperties(java.util.Properties)`: Called before `parse()` by the DigitalEdge framework. Save any properties associated with the InputStream.

Abstract classes

Several Abstract classes that partially implement the `Parser` interface are provided by the SDK and recommended for use when writing your own parser class. They are:

`AbstractBufferingParser`: Abstract base class for parsers that may want to return more than one output record from a single parsed input record. Extending classes must override one of the two parse methods.

`AbstractLineParser`: Abstract base class for parsers that need to interpret data as lines of text. This class assumes that the input is new line terminated text. Methods are provided to extract input from a stream one line at a time as well as to split delimited data into fields.

`AbstractXMLParser`: Abstract base class for parsers that need to interpret data as XML.

(See the *Ingest SDK Javadocs* for more detailed information on these classes.)

SimpleLogParser.java

```
package <your package>;

import java.text.ParseException;
import java.util.HashMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import net.sf.json.JSONObject;

import org.apache.log4j.Logger;

import com.saic.rtws.core.framework.parser.AbstractLineParser;

public class SimpleLogParser extends AbstractLineParser {

	private static final Logger log = Logger.getLogger(SimpleLogParser.class);

	//Timestamp format: "YYYY-MM-DD HH:MM:SS" or "WWW MMM DD HH:MM:SS" or "MMM DD HH:MM:SS" or "DD/MMM/YYYY:HH:MM:SS"
	//or "HH:MM:SS"
	protected static final String TIMESTAMP_REGEX =
		"((([0-9]{1,2}/[A-z]{3}/[0-9]{4}:)|([0-9]{4}-[0-9]{2}-[0-9]{2} *)|(([A-z]{3} *)+[0-9]{1,2} ))
		*[0-9]{2}:[0-9]{2}:[0-9]{2})";
	protected static final Pattern TIMESTAMP_PATTERN = Pattern.compile(TIMESTAMP_REGEX);

	protected String defaultSource;

	protected String defaultAccessLabel;

	public SimpleLogParser() {
		super("UTF-8");
	}

	@Override
	public void parseHeaders() {
	}
```

```
        public JSONObject parse() {
                try {
                        String input = nextRecord();
                        if (input == null) return null;

                        String streamAccessLabel = null; //info.getProperty(StandardHeader.ACCESS_LABEL_KEY);
                        String accessLabel = (streamAccessLabel == null) ? defaultAccessLabel : streamAccessLabel;

                        String streamSource = null; //info.getProperty(StandardHeader.SOURCE_KEY);
                        String source = (streamSource == null) ? defaultSource : streamSource;
                        HashMap<String, String> map = new HashMap<String, String>();
                        Matcher m;

                        //Step One: Process Timestamp
                        m = TIMESTAMP_PATTERN.matcher(input);
                        if(!m.find())
                                throw new ParseException("No timestamp found.", -1);
                        else
                                map.put("TIMESTAMP", m.group());

                        //Cut the timestamp out of the input so it doesn't get in the way
                        input = input.replaceFirst(TIMESTAMP_REGEX, "");

                        //Step two: Process Message (the rest of the input)
                        map.put("MESSAGE", input);

                        JSONObject json = new JSONObject();
                        json.element("TIMESTAMP", map.get("TIMESTAMP"));
                        json.element("MESSAGE", map.get("MESSAGE"));
                        return json;

                } catch (ParseException e) {
                        log.error("Error parsing record in ConfigurableLogParser: ", e);
                        return null;
                }
        }
}
```

Creating an enrichment

Enrichments add to the data entering your system, utilizing methods such as enhancing data through calculations or mapping it to other data dimensions. Enrichments can be chained, with one enrichment enhancing the output of a previous enrichment. Enrichments are data-specific and must be developed specifically for your data. This is not to say that enrichments cannot be used across different systems; but the data entering your system must have the correct fields and configuration for the enrichment to work. Use the DigitalEdge SDK to write custom enrichments that can be plugged in to the DigitalEdge framework and made available through the DigitalEdge setup tools.

To create an enrichment, you must extend the `com.saic.rtws.core.framework.processor.AbstractEnrichmentProcessor` class. The key methods for this class are described below.

AbstractEnrichmentProcessor

`buildEnrichedElement()`: This method is the workhorse of the class. It is the method called to perform the actual enrichment of the data. The return type of this method is `java.lang.Object`, and the value to be returned is the actual enrichment for the data. The base class of `AbstractEnrichmentProcessor` takes this value and handle, inserting it back into the data record.

`getType()`: This method must be implemented by all classes extending `AbstractEnrichmentProcessor`. It declares the type of enrichment performed by the processor. It simply returns a `java.lang.String` that is the processor's type, i.e. a NOOPEnrichment would return "noop".

The `AbstractEnrichmentProcessor` class also implements the `com.saic.rtws.common.util.initializable` interface as outlined below.

Initializable

`initialize()`: Called to set up the enrichment before use. Used to create the connection, initialize resources, etc.

`dispose()`: Called when use of the enrichment is complete. Used to clean up connections and resources created in `initialize()`.

SimpleAreaCalculationEnrichment.java

```
package <your package>;

import net.sf.json.JSONObject;

import com.saic.rtws.core.framework.processor.AbstractEnrichmentProcessor;
import com.saic.rtws.core.framework.processor.EnrichmentAction;
import com.saic.rtws.core.framework.processor.ParameterList;

/**
 * Extremely simple example Enrichment Processor
 */
public class SimpleAreaCalculationEnrichment extends AbstractEnrichmentProcessor {

	@Override
	public Object buildEnrichedElement(EnrichmentAction action, ParameterList parameters) {

		/* Asumming a json record such as
		{
		    "rectangle": {
		        "length": "10",
		        "width": "20"
		    }
		}
		This enrichment simply retrieves the length and width of the rectangle and calculates the area.
		The process of extracting the input parameters from and inserting the result back into the data
		record is handled by @see AbstractEnrichmentProcessor
		*/

		JSONObject obj = action.getRootObject();
		Integer length = obj.getJSONObject("rectangle").getInt("length");
		Integer width = obj.getJSONObject("rectangle").getInt("width");
		Integer area = length * width;


		return area;
	}
```

```
        @Override
        public String getType() {
                return "AreaCalculation";
        }

}
```

Creating a data sink

Data sinks receive pipeline-processed data for storage, alerting, or post-processing. DigitalEdge comes with many build-in user-configurable data sinks such as HBase, internal or external Hive, Lucene, MongoDB, and Alerting. If the built-in data sinks do not meet your needs, use the DigitalEdge SDK to write a custom data sink that can be plugged into the DigitalEdge framework and made available through the DigitalEdge setup tools.

To create a data sink, you must write a class that extends the `com.saic.rtws.core.framework.processor.AbstractDataSink` class. The `AbstractDataSink` class has a public subclass of `FlushCounter`. The key methods for these classes are described below.

AbstractDataSink

`processInternal(net.sf.json.JSONObject)`: Called to process a JSON object and store it in the data sink. This method is meant for internal use only; any implementation of the data sink should call `AbstractProcessor process(netsf.json.JSONOBJECT)` which is inherited as described below.

FlushCounter

Methods count records and bytes processed, and verify the accuracy of the count.

The `AbstractDataSink` class extends the `com.saic.rtws.core.framework.AbstractProcessor`, implements the `com.saic.rtws.core.framework.DataSink` interface, and through inheritance also implements the `com.saic.rtws.common.util.initializable` interface as outlined below.

AbstractProcessor

`process(net.sf.json.JSONObject)`: Called to process a record that comes in the format of a JSON object. This is the method that should be invoked to process records, not `AbstractDataSink.processInternal(net.sf.json.JSONObject)`, as it provides a check to make sure the processor is enabled and the event is eligible for processing.

DataSink

`flush()`: Called to flush any cached records to the data sink. This method can be configured to flush after a specified amount of bytes, records, or time between records, via other methods of the `DataSink` interface.

Initializable

`initialize()`: Called to set the data sink up before use. Used to create the connection to the data sink, initialize resources, etc.

`dispose()`: Called when use of the data sink is complete. Used to clean up connections and resources created in `initialize()`.

The following sample is a simple text file data sink. It saves all records received by the system into the text file:

```
/usr/local/etc/datasinkFile.txt
```

in a JSON string format. In addition to the code for the data sink, the `JsonFileDataSinkTest.java` sample is a test driver for testing the data sink locally. The test driver runs locally as a JUnit test.

To integrate a data sink with DigitalEdge, contact a DigitalEdge Support Engineer to assign it to a process group.

JsonFileDataSink.java

```
package <your package>;

import java.io.File;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

import net.sf.json.JSONObject;

import com.saic.rtws.commons.exception.InitializationException;
import com.saic.rtws.core.framework.DataSink;
import com.saic.rtws.core.framework.Description;

@Description("This is a data sink for an Json File Data Sink.")
public class JsonFileDataSink extends com.saic.rtws.core.framework.processor.AbstractDataSink implements DataSink {

	final private String DATA_SINK_FILE_PATH = "/usr/local/etc";		// Location of the Data Sink File
	final private String DATA_SINK_FILE = "datasinkFile.txt";		// Data Sink File name
	private BufferedWriter dataSinkOut;					// A writer to the Data Sink File

	/** Constructor. */
	public JsonFileDataSink() {
		super();
	}

	/**
	 * Perform disposal work for the data sink.
	 *
	 * In our case, this means closing the Data Sink File.
	 */
	public void dispose() {
		// Close Connection to Data Sink File
		try {
			System.out.println("DEBUG: About to close the data sink file");
			dataSinkOut.close();
			System.out.println("DEBUG: Data sink file successfully closed");
```

```
                } catch (IOException e) {
                        System.out.println("ERROR: Data sink file failed to close: " + e.getMessage());
                }

                return;
        }

        /**
         * Perform initialization work for the data sink.
         *
         * In our case, we need to make sure the Data Sink File path exists
         * and then open the file.  If an error occurs with either of these
         * we have failed to initialize and therefore need to throw an
         * InitializationException.
         *
         * @throws InitializationException Could throw an InitializationException
         */
        public void initialize() throws InitializationException {

                // Make sure the file path exists
                if(!(new File(DATA_SINK_FILE_PATH).exists())) {
                        // The file path does not exist so we must create it.
                        System.out.println("DEBUG: " + DATA_SINK_FILE_PATH + " does not exist. Creating directories");
                        if(!new File(DATA_SINK_FILE_PATH).mkdirs()) {
                                // The path failed to be created so throw an InitializationException
                                throw new InitializationException(DATA_SINK_FILE_PATH + " was not successfully created");
                        }
                        System.out.println("DEBUG: " + DATA_SINK_FILE_PATH + " successfully created");
                }
                else {
                        // The file path exists, continue on
                        System.out.println("DEBUG: " + DATA_SINK_FILE_PATH + " exists");
                }

                // Open the connection to Data Sink File
                try {
                        System.out.println("DEBUG: About to open the data sink file");
                        dataSinkOut = new BufferedWriter(new FileWriter(DATA_SINK_FILE_PATH + DATA_SINK_FILE));
                        System.out.println("DEBUG: Data sink file successfully opened");
                } catch (IOException e) {
```

```
                        // File failed to open so throw an InitializationException
                        throw new InitializationException("Data sink file failed to open: " + e.getMessage(), e);
                }

                return;
        }

        /**
         * Process the record to be stored in the data sink.
         *
         * In our case, this means converting the record into text format for the
         * Data Sink File.
         */
        protected void processInternal(JSONObject record, FlushCounter counter) {
                // We want to count the record as received.  AbstractData sink has a
                // FlushCounter in it which counts both the number of records received
                // as well as the number of bytes received.  This is useful when using
                // data sinks which require caching before flushing to the data sink.
                // In our case, we do not cache the records before flushing to the
                // data sink, but we can still track the amount received.
                counter.increment(1, record.toString().length());

                // Write the record directly to the Data Sink File
                try {
                        System.out.println("DEBUG: About to write record \"" + record.toString() + "\" to data sink file");
                        dataSinkOut.write(record.toString() + "");
                        System.out.println("DEBUG: Record successfully written to data sink file");
                } catch (IOException e) {
                        // Writing to file failed so log the error
                        System.out.println("ERROR: Failed to write record to data sink file: " + e.getMessage());
                }
        }

        /**
         * Perform flush for the data sink.
         */
        public void flush() {
                // In our case, there is no caching to flush.

                // A data sink can be configured to flush based on any combination
```

```
                // of the number of records between flushes, number of
                // bytes between flushes, or amount of time(ms) between flushes.
                // If none of these are configured, the flush must be called explicitly.
                System.out.println("DEBUG: Data sink flush() called erroneously");
        }

}
```

JsonFileDataSinkTest.java

```
package <your package>;

import java.io.BufferedReader;
import java.io.FileReader;
import junit.framework.TestCase;
import net.sf.json.JSONObject;
import net.sf.json.JSONSerializer;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import com.saic.rtws.commons.exception.InitializationException;

/**
 * A driver class for testing the JsonDataSink.
 *
 * The class is in the proper format to be run as a JUnit test.
 */
public class JsonFileDataSinkTest extends TestCase {

	private static JsonFileDataSink dataSink = new JsonFileDataSink();	// An instance of the data sink
	private String baseDir;							// A string to the user's base directory
	private boolean safeToTest = false;			// A flag to determine whether the test is safe to run

	/**
	 * Ran to clean up after the JUnit test.  In this example, we wnat to dispose of the data sink
	 */
	@After
	public void cleanUp() {
		// Clean up the data sink
		dataSink.dispose();
	}

	/**
	 * Ran to set up before the JUnit test.  In this example, we need to choose the find the
	 * base directory since that is where we have the input data file stored.  In addition,
	 * we need to initialize the data sink.
```

```
 */
@Before
public void setUp() {
        // Get the user's base directory
        if (System.getProperty("basedir") == null) {
                System.setProperty("basedir", System.getProperty("user.dir"));
        }

        baseDir = System.getProperty("basedir");

        if (baseDir == null) {
                System.out.println("ERROR: Need to set basedir in system properties at startup time");
        }
        else {
                baseDir = System.getProperty("user.dir");
        }
        System.out.println("INFO: baseDir = " + baseDir);

        // Initialize the data sink
        try {
                System.out.println("INFO: Initializing data sink");
                dataSink.initialize();
                safeToTest = true;
                System.out.println("INFO: Data sink successfully initialized");
        }
        catch(InitializationException e) {
                // The data sink failed to initialize, flag that we do
                // not want to continue testing
                safeToTest = false;
                System.out.println("ERROR: " + e.getMessage());
        }
}

@Test
public void testProcessInternal() throws Exception {
        // Check if the test is safe to run or if the data sink failed to initialize
        if(safeToTest) {
                BufferedReader testDataIn = null;

                try {
```

```
                                // Open the file to read the input data from.  This file
                                // is just text file with a JSON string on each line.
                                System.out.println("INFO: new dir:");
                                testDataIn = new BufferedReader(new FileReader(baseDir
                                                + "/src/resource/test.json"));

                                // Work through the file reading each line of text
                                String line;
                                while ((line = testDataIn.readLine()) != null) {
                                        // Convert each line of text into a JSONObject record
                                        JSONObject record = (JSONObject) JSONSerializer.toJSON(line);
                                        System.out.println("INFO: Read record \"" + record + "\"");

                                        // Process the the record. Notice here that we do not call the
                                        // processInternal(JSONObject record, FlushCounter counter) function
                                        // of the Data Sink.  Instead we just call process(JSONObject record)
                                        // function.  That is because the processInternal is only used internally
                                        // by the data sink in conjunction with the data sink's FlushCounter.
                                        dataSink.process(record);
                                        System.out.println("INFO: Record written to data sink");
                                }
                        } finally {
                                // Close the test data file
                                testDataIn.close();
                        }
                }
                else {
                        // Log that the test did not run
                        System.out.println("INFO: testProcessInternal did not run since data sink failed to
     initialize");
                }
          }
}
```

How to Run the Samples

To facilitate the development of custom plug-ins, Apache Maven™ archetypes are available for download and use. These archetypes should be used as the starting point for any custom plug-ins, as they provide a complete build environment for development. To use and run these samples, follow these steps:

1. Download the desired archetype and its corresponding POM file.

a. In the **Management Console**, select the **Plug-ins** section.

b. On the **Software** tab, click the help icon for **Getting Started with Software Plug-ins**.

c. Click the **Sample Projects** link to access the archetypes directory. Select and save the desired file(s).

2. Once downloaded, install the archetype in your local Maven repository:

```
mvn install:install-file -Dfile=<path to the downloaded archetype> \
        -DgroupId=com.saic \
        -DartifactId=<plugin type>-example-archetype \
        -Dversion=<version of the archetype downloaded> \
        -Dpackaging=jar \
        -DpomFile=<plugin type pom downloaded>
```

If you are using a Maven repository manager (e.g. Sonatype Nexus™, Apache Archiva™), you can upload it to the internal repository manager to avoid this local install.

3. When all of the desired plug-in archetypes are installed:

a. If you are using a local repository install approach, issue the following command to update the local archetype catalog:

```
mvn archetype:crawl
```

b. If the local archetype-catalog is not created/updated, add the following stanza to your local Maven repository's `archetype-catalog.xml`

```
<archetype>
<groupId>com.saic</groupId>
<artifactId>parser-example-archetype</artifactId>
<version>1.2.1</version>
</archetype>
```

c. The content of the `C:\Users\<your username>\.m2\archetype-catalog.xml` should look similar to the following (if using the `parser-example-archetype`):

```
<?xml version="1.0" encoding="UTF-8"?>
<archetype-catalog xsi:schemaLocation="<http://maven.apache.org/plugins/
   maven-archetype-plugin/archetype-catalog/1.0.0 http://maven.apache.org/
   xsd/archetype-catalog-1.0.0.xsd>"
       xmlns="http://maven.apache.org/plugins/maven-archetype-plugin/
```

```
          archetype-catalog/1.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
       <archetypes>
              <archetype>
                     <groupId>com.saic</groupId>
                     <artifactId>parser-example-archetype</artifactId>
                     <version>1.2.1</version>
                     <description>parser-example-archetype</description>
              </archetype>
       </archetypes>
</archetype-catalog>
```

4. To use an archetype, issue the following command to produce a runnable plug-in project for custom development (this example uses the `parser-example-archetype` to create a parser plug-in project):

```
mvn archetype:generate -DarchetypeCatalog=local -DarchetypeGroupId=com.saic \
        -DarchetypeArtifactId=parser-example-archetype -DarchetypeVersion=<version>
```

You will be prompted for your plug-ins project's group id, artifact id, and version.

5. Once your project is created, you can review and develop the custom plug-in using an editor of your choice. To build/test the custom plug-in, execute the written JUnit tests or execute the following command from your plug-in's project folder:

```
mvn clean test
```

Programming and Testing Environment

To test a component in-line with DigitalEdge, a plug-in must be installed with your system. Currently, transports, parsers, and enrichments can be uploaded to the repository with the **Plug-ins** feature of DigitalEdge's **Management Console** and accessed via **System Builder** to assemble a processing pipeline. See the *Operations Guide* for details.

To integrate a data sink with DigitalEdge, contact a DigitalEdge Support Engineer to assign it to a process group.

Tips and tricks

- Each private component must be packaged in a jar file, should contain a pom.xml file with the plug-in's dependencies described in the Maven dependency format, and should not contain nested jars. If you use the provided plug-in archetypes as a starting point, these requirements are all met. For more information about Maven and its capabilities, see http://maven.apache.org/.
- After you create a custom plug-in, you need to upload the component into the TMS Master Repository from your hard drive. Use the **Plug-ins** section in the DigitalEdge **Management Console** to upload components. See the *Operations Guide* for details.
- Once the component is in the Master Repository, you can use it when building a DigitalEdge system. Access parsers and enrichments when creating an input model in **Data Model Editor**; access transports and data sinks when assembling a processing pipeline in **System Builder**.

Index