EasySMF:JE Installation and User Guide

Version 2.0

EasySMF:JE Installation and User Guide

Table of Contents

Introduction

EasySMF:JE provides a powerful and intuitive API for processing SMF data.

EasySMF:JE runs on z/OS and other Java platforms, and allows you to use the advanced functions of Java for your SMF reporting. It is 100% Java code so is zIIP eligible on z/OS.

Processing SMF Data with Java

Java is a highly optimized language which runs very fast on z/OS. The object-oriented structure of Java classes are very well suited for representing SMF records and sections. This makes Java an excellent choice for processing SMF data.

Java will automatically make use of features like SIMD on the latest z/OS systems to increase the speed of the Java code.

However, the biggest improvements in speed don't come from writing faster code, they come from using more efficient algorithms. This is where Java has a huge advantage over more traditional languages used for SMF processing.

The **Java Collections** classes provide the foundation for more efficient algorithms. In particular, the **java.util.HashMap<>** class allows very efficient processing. Sorting input data is not necessary, which can be a big overhead for SMF reporting. Many of the EasySMF samples demonstrate use the HashMap to organize the data for a report.

EasySMF:JE Concepts

EasySMF:JE is designed to provide a consistent API across different record types. Names are based on the names of the SMF fields, using Java naming conventions e.g. SMF30CPT becomes smf30cpt().

You will need to refer to the various product documentation for the meanings of the SMF fields.

Reading Records

The **com.blackhillsoftware.SmfRecordReader** class reads records from a SMF file or DDNAME.

On z/OS you will normally read from a SMF dump file allocated to a DDNAME in your job using the JZOS Batch Launcher.

SmfRecordReader can also read from files on Linux/Windows or z/OS OMVS. These files must have been transferred as BINARY, and include the record descriptor words (RDWs) which contain the record length information. SMFRecordReader can also read VBS files transferred with RECFM=U, i.e. for use with other SMF processing tools on distributed platforms.

Reading Sections from a Record

Records and sections provide methods to extract sections and subsections.

- Subsection and method names are based on the description in the SMF documentation, e.g. **com.blackhillsoftware.smf.smf30.PerformanceSection**.

- Where multiple sections may be present the sections are returned as a **List<SectionType>**. This applies to most sections described by triplets (offset, length, count) as well as other instances where there may be multiple sections, e.g. chained using pointers to the next section. This means that you do not have to deal with the logic to extract the sections yourself.
- If there are no sections present, e.g. triplets with a count of 0, an empty List<> is returned. This allows you to use iterators to access the sections without explicitly testing whether any are present - if there are none you will simply iterate 0 times.
- If it is documented that there will always be exactly one section, the method will return that section.
- If there can be one or 0 sections a List<SectionType> containing 1 or 0 sections will be returned. There may also be an alternate method which returns the section if present, or null.

Examples

SMF type 30 records always have 1 Identification section so the **Smf30Record** class has a method called **identificationSection()** that returns a class called IdentificationSection.

```
IdentificationSection ident = record.identificationSection();
```

SMF type 30 records can have 1 or 0 Completion sections so the **completionSections()** method returns a List<CompletionSection> that will have 0 or 1 entry.

```
List<CompletionSection> comp = record.completionSections();
```

It is documented that there can be no more than one completion section, so there is also a **completionSection()** method which provides a single completion section. It returns null if there is no completion section present.

```
CompletionSection comp = record.completionSection(); // might return null
```

SMF type 30 records can have multiple EXCP sections so the excpSections() method returns a List<ExcpSection> that will have zreo or more entries.

```
List<ExcpSection> excp = record.excpSections();
```

Data Types

SMF data types are converted to Java data types as follows:

- **1, 2 and 3 byte integer** values are converted to the Java **int** datatype (32 bits). Java integers are signed, so 32 bits is too small for a 4 byte/32 bit unsigned value.
- **4 byte/32 bit unsigned integer values** are converted to a Java **long** (64 bits).
- **8 byte/64 bit unsigned integer** values are available as both **long** and **BigInteger** values. The long data type may provide better performance if the value will not exceed the maximum 64 bit signed value. The API will throw an exception if the value is too large for a long. If this is possible, use the BigInteger value.
- **Integers greater than 8 bytes** are converted to **BigInteger**s.
- **Floating point** values are converted to Java **double**s.

Dates and Times

Dates and times are represented using **java.time** classes. The java.time classes provide nanosecond resolution which covers most of the date and time values in SMF data. The unconverted, raw values for each field are also available if required.

- Times representing a duration e.g. CPU time, connect time etc. are converted to **java.time.Duration** values. They are also available as a **double** value in seconds, which is likely to be more convenient with better performance for arithmetic. E.g. The SMF 30 ProcessorAccountingSection provides the SMF30CPT value as a Duration from **smf30cpt()** and a double value in seconds in **smf30cptSeconds()**.

- Dates and times of day are represented as java.time classes - **LocalDate**, **LocalTime**, **LocalDateTime**, **ZonedDateTime** depending on which components of the time are included - date, time, timezone etc. The different classes help avoid errors caused by comparing fields which are not equivalent e.g. ZonedDateTime and LocalDateTime.

- Times representing offsets from GMT are returned as **java.time.ZoneOffset**. This can be combined with a LocalDateTime to create a ZonedDateTime.

- STCK values are returned as a **ZonedDateTime**. The actual STCK value is also available as a BigInteger if the full precision of STCK is required.

Incorrect Data Types

If you find fields where the data type is incorrect (i.e. doesn't follow these principles), please report it to support@blackhillsoftware.com. Fixing it is likely to be a "breaking change" and require changes and/or recompilation for any programs using that field, so it is better to report it and get it fixed than code around it and have someone else report it later.

Installation on z/OS

Prerequisites

Java 8 or higher

EasySMF:JE runs under Java 8 or higher.

JZOS Batch Launcher

EasySMF:JE can run under the OMVS shell, BPXBATCH or using the JZOS Batch Launcher. The JZOS Batch Launcher is recommended because it allows you to use DD names in JCL as for any other z/OS batch job.

Install the JZOS Batch launcher according to instructions in the IBM *JZOS Toolkit Installation and Users Guide.*

This consists of the following steps:
1. Copy the JVMLDM80 and JVMLDM86 load modules to a PDSE.
2. Customize the JVMPRC80 (32 bit) and JVMPRC86 (64 bit) JCL procedures.

Copy Installation

EasySMF:JE consists of a single jar file plus dependencies.

The easiest way to use EasySMF:JE is to build programs on a Windows or Linux platform using Apache Maven and transfer the resulting jar files to z/OS. Sample projects to build reports using EasySMF:JE are available on Github:
https://github.com/BlackHillSoftware/easysmf-samples

The sample projects copy dependencies including the EasySMF:JE jar to the `target/lib` directory. Transfer the build output from `target/*.jar` and `target/lib/*.jar` to corresponding directories on z/OS using binary transfer options. JCL to run the programs is provided in the Github repository:
https://github.com/BlackHillSoftware/easysmf-samples/tree/main/JCL

z/OS Installation

EasySMF:JE version 2 is supplied as a pax archive containing the EasySMF:JE jar, samples, and dependencies. Source to the sample programs and JCL is also provided.

1. Transfer the pax archive to a direcory on z/OS using binary transfer options.
2. Select a location to install EasySMF:JE.

   For testing you can install into your home directory. The archive will be extracted into a subdirectory named `easysmf-je-v.r.m` where v.r.m is the version information.

   Several subdirectories and files will be created under the easysmf-je-v.r.m directory:

`jar`	Contains the EasySMF and related jar files required to run EasySMF.
`samples`	EasySMF sample reports
`samples/jar`	Source code for the EasySMF sample reports
`samples/jar/lib`	Additional dependencies for the sample programs

`samples/JCL`	JCL to compile and run Java programs
`samples/scripts`	Scripts and Windows batch files to compile and run Java programs
`samples/easysmf-skeleton`	A skeleton Maven project to build a reporting program using EasySMF:JE. Copy and modify this skeleton to create your own programs.
`samples/sample-reports`	A selection of sample reports
`samples/smf-de-dup`	A program to remove duplicate records from SMF data.
`samples/smf-report-dups`	A program to analyze duplicate records in SMF data.
`samples/smf2json`	Sample programs to produce JSON from SMF data
`samples/smf2json-skeleton`	A skeleton Maven project to convert SMF data to JSON using EasySMF:JE. Copy and modify this skeleton to create your own programs.
`samples/tutorial`	The EasySMF:JE tutorial from https://github.com/BlackHillSoftware/easysmf-samples/tree/main/tutorial

The samples/ directory is a copy of the EasySMF Samples project on Github:
https://github.com/BlackHillSoftware/easysmf-samples

3. Extract the pax archive:
```
pax -rvf easysmf-je-v.r.m.pax
```

4. Install the license or temporary key.

   On z/OS, you can use DD name EZSMFKEY or the EASYSMFKEY environment variable to indicate the location of the EasySMF:JE license key.

   EZSMFKEY DD name:

```
//EZSMFKEY DD DISP=SHR,DSN=YOUR.DATASET(EZSMFK)
```

   If you use the EASYSMFKEY environment variable the key itself can be installed in a file or z/OS dataset. z/OS datasets are indicated by names beginning with //.
   Quotes and brackets in the dataset name need to be escaped using backslash characters

   The following example shows how to point the EASYSMFKEY environment variable to a member of a PDS.

```
//STDENV DD *
export EASYSMFKEY=//\'YOUR.DATASET\(EZSMFK\)\'
```

   The EASYSMFKEY environment variable could also be added to the standard environment via `/etc/profile` or `.profile`.

   See topic EasySMF:JE Key for more information.

5. IVP

   JCL from `samples/JCL/RUNJZOS.jcl` can be used to verify the installation.
   Update the jobname and EZSMFDIR, JZOSLIB, SMFDATA symbols as required. Submit the job and check it runs successfully.

   Common Problems

```
JVMJZBL2008E Could not find or load class: com.smfreports.json.Smf30RecordToJson
```

Double check the directories used for the class path in the system symbols. If there is an error in the name the non-existent directory is added to the CLASSPATH without any error message, and Java fails when it tries to locate the classes.

Java Class Path

The following jars are distributed with EasySMF and need to be included in the class path for your own EasySMF reporting programs:

`easysmf-je-`*`v.r.m`*`.jar`
`slf4j-api-`*`v.r.m`*`.jar` - Simple Logging Facade for Java. This allows the destination for EasySMF messages to be customized.
`slf4j-simple-`*`v.r.m`*`.jar` - Directs EasySMF messages to stderr. This jar can be replaced with an alternative SLF4J binding to customize logging behavior. See SLF4J Message Logging.

Installation on other platforms

Prerequisites

Java 8 or higher

EasySMF:JE runs under Java 8 or higher.

Apache Maven

Apache Maven is recommended to build the projects and manage dependencies.
https://maven.apache.org/

Building using Apache Maven

1. Clone the EasySMF Samples project from Github, or download and extract as a zip file
   `git clone https://github.com/BlackHillSoftware/easysmf-samples.git`
   or
   download from https://github.com/BlackHillSoftware/easysmf-samples/archive/refs/heads/main.zip

2. Build the samples e.g.
   ```
   cd easysmf-samples/sample-reports
   mvn clean package
   ```
   The first time you run Maven it needs to download all the plugins used in the build, plus the project dependencies. This can be a lengthy list of downloads. These are cached locally on your machine so they do not have to be downloaded for subsequent builds.
   The output jar file is written to the `target` subdirectory. The supplied Maven pom.xml uses the Maven Dependency Plugin to copy dependencies to the `target/lib` directory.

3. Install the license or temporary key.
   The EASYSMFKEY environment variable points to the file containing the EasySMF:JE license key. The key needs to be saved into a file and the EASYSMFKEY environment variable created to point to the fie.
   See topic EasySMF:JE Key for more information.

4. Run the program.
   Sample unix scripts and Windows batch files are provided in the samples `scripts` subdirectory.
   Set the TARGET variable to the `target` directory with the output from the build.
   Set the EASYSMFKEY variable to point to the file with the temporary or permanent key.
   The EASYSMFLOCATION variable is not important for a Maven build, because the dependencies including EasyS

5. Run on z/OS if required.
   a. Copy the jar files from the `target` and `target/lib` directories to z/OS using binary transfer options.
   b. JCL to run on z/OS is in the samples `JCL` subdirectory.
      See the Installation on z/OS topic for information on running under the JZOS Batch Launcher.

Zip or tar.gz Archive

EasySMF:JE is also available as zip and tar.gz archives for installation on non-z/OS systems. The content is the same as the pax archive included in the z/OS installation.

1. Extract the files from the zip or tar.gz archive.

2. Install the license or temporary key.
   The EASYSMFKEY environment variable points to the file containing the EasySMF:JE license key. The key needs to be saved into a file and the EASYSMFKEY environment variable created to point to the file.
   See topic EasySMF:JE Key for more information.

Compile

Compile scripts are located in the `samples/scripts` subdirectory.

Update the scripts:
1. Set the EASYSMFLOCATION variable to the location of the extracted files.
2. Set the TARGET variable to the output directory for the compiled class files.
3. Run the compile script e.g.
   ```
   ./compilesample.sh sample-reports/src/main/java/com/smfreports/RecordCount.java
   ```

Run

Run scripts are also located in the `samples/scripts` subdirectory.

1. Set the EASYSMFLOCATION variable to the location of the extracted files.
2. Set the TARGET variable to the output directory from the compilation step.
3. Set the EASYSMFKEY variable to point to the file with the temporary or permanent key.
4. Run the script e.g.
   ```
   ./runsample.sh com.smfreports.RecordCount SMF.DATA
   ```

Java Class Path

The following jars are distributed with EasySMF and need to be included in the class path for your own EasySMF reporting programs:

`easysmf-je-v.r.m.jar`
`slf4j-api-v.r.m.jar` - Simple Logging Facade for Java. This allows the destination for EasySMF messages to be customized.
`slf4j-simple-v.r.m.jar` - Directs EasySMF messages to stderr. This jar can be replaced with an alternative SLF4J binding to customize logging behavior. See SLF4J Message Logging.

Creating your own reports

Skeleton projects are provided with the EasySMF:JE samples as a basis for your own reporting projects.

- **easysmf-skeleton** is a simple project to build an EasySMF:JE reporting program
- **smf2json-skeleton** is a project to build a program to convert SMF data to JSON format using the EasySMF-JSON functions.

Start by editing the project `pom.xml` file and changing the **groupId**, **artifactId** and **name** elements as required.

Use an IDE such as Eclipse or Visual Studio Code and import the Maven project into the Workspace. Make modifications as required.

Change the **mainClass** element in the pom.xml file to reflect your main class name.

Build the project using maven. Change to the directory containing pom.xml and enter the command:

```
mvn clean package
```

The skeleton projects use the maven-dependency-plugin to copy the project dependencies to the `target/lib` output directory. The run time class path needs to include the jars from the `target` and `target/lib` directories.

If you create a runnable jar (using the mainClass element in the pom) the class path is set at build time, and the dependencies must be in the directory specified by the pom **classpathPrefix** i.e. `./lib` in the distributed samples.

EasySMF:JE Key

The EasySMF:JE key is a text file that looks like:

```
**License:
MQ0KMjAxNS0wOS0wNA0KVGVtcG9yYXJ5IEtleQ0K
**Sig:
bBvA9pCvG145BQrvaser65geM8zaIxJsAt5XJVtwpB3Ld4TXB4LzaCFNdRgXtQQR
IGAg+KhMLyaMM66gFMsFQduE1A5AK6wWi2Z9hBBm/YOBqCW4gbw9iWzeApE09VaQ
LbFkhc6WNyv2/27VZXJ/nkAyUBzX9uyVEKb4Vc+oX0M=
**End
```

Include all lines, including the **License and **End lines. The format of the key is designed so that it can be easily pasted into a 3270 emulator session.

EasySMF locates the key via a DD name on z/OS, or using the EASYSMFKEY environment variable on z/OS or other platforms.

In a z/OS batch job you can simply add a DD in the JCL:

```
//EZSMFKEY DD *
**License:
MQ0KMjAxNS0wOS0wNA0KVGVtcG9yYXJ5IEtleQ0K
**Sig:
bBvA9pCvG145BQrvaser65geM8zaIxJsAt5XJVtwpB3Ld4TXB4LzaCFNdRgXtQQR
IGAg+KhMLyaMM66gFMsFQduE1A5AK6wWi2Z9hBBm/YOBqCW4gbw9iWzeApE09VaQ
LbFkhc6WNyv2/27VZXJ/nkAyUBzX9uyVEKb4Vc+oX0M=
**End
/*
```

In other environments, including z/OS unix, the key is located using the EASYSMFKEY environment variable.

Save the key to a file, and set the environment variable to the name of the file.

On Windows:
`set "EASYSMFKEY=C:\path to your\key.txt"`
or got to Control Panel -> Edit the System Environment Variables and add the variable.

On unix:
`export EASYSMFKEY="/etc/easysmf/easysmfkey.txt"`

30 Day Trial

To obtain a key for a 30 day trial of EasySMF:JE, visit:
https://www.blackhillsoftware.com/30-day-trial/

Reading SMF data

Reading SMF data on z/OS

EasySMF:JE uses the **SmfRecordReader** class to read data from RECFM=VB or RECFM=VBS SMF dump datasets.

The JZOS Batch launcher allows you to run Java programs with access to DD statements defined in JCL. Allocate the SMF dump dataset to a DDNAME in your job, and open and read it using the SmfRecordReader class.

The SmfRecordReader class implements the **Iterable<SmfRecord>** and **Closeable** interfaces so you can iterate to read each record, and use "try with resources" to automatically close the file.

```
try (SmfRecordReader reader = SmfRecordReader.fromDD("INPUT"))
{
    for (SmfRecord record : reader)
    {
        // Process each record here
    }
}
// reader is automatically closed when exiting the try block
```

SmfRecordReader can also access DD names and MVS datasets using SmfRecordReader.fromName(...). This means programs can be run on z/OS or other platforms simply by passing names in different formats to the program.

```
try (SmfRecordReader reader = SmfRecordReader.fromName("//DD:INPUT"))
{
    for (SmfRecord record : reader)
    {
        // Process each record here
    }
}
```

or to access a z/OS dataset by name without preallocating a DD:

```
try (SmfRecordReader reader = SmfRecordReader.fromName("//'MVS.DATASET.NAME'"))
{
    for (SmfRecord record : reader)
    {
        // Process each record here
    }
}
```

If you provide the dataset name or DDNAME in the program arguments SmfRecordReader.fromName(...) allows you to use the same code to read from a dataset name or DDNAME on z/OS, or a file name on other platforms.

```
try (SmfRecordReader reader = SmfRecordReader.fromName(args[0]))
{
    for (SmfRecord record : reader)
    {
```

```
            // Process each record here
        }
    }
```

Reading SMF data on other platforms

Reading on other platforms is very similar to reading on z/OS, except that the SmfRecordReader reads from a file or some other form of InputStream.

The stream data must include the record descriptor words (RDWs) so that the record lengths can be determined. Optionally it can include the block descriptor word (BDW) i.e. if a VB or VBS file was transferred as RECFM=U.

This example reads from a file, where the filename is passed as an argument to the program.

```
    try (SmfRecordReader reader =
        SmfRecordReader.fromName(args[0]))
    {
        for (SmfRecord record : reader)
        {
            // Process each record here
        }
    }
```

SmfRecordReader.fromName(...) opens a FileInputStream using the argument as the name of the SMF data file.

Reading SMF data from a stream

SMF data can also be read from an InputStream. This allows a variety of sources, e.g. a TCP/IP network stream, or passing data through a stream to compress and decompress data.

The stream data must include the record descriptor words (RDWs) so that the record lengths can be determined.

This example reads from a FileInputStream created from a file name passed as an argument to the program.

```
    try (SmfRecordReader reader =
        SmfRecordReader.fromStream(new FileInputStream(args[0])))
    {
        for (SmfRecord record : reader)
        {
            // Process each record here
        }
    }
```

Selecting Record Types for Processing

The SmfRecordReader class can filter records by type and subtype.

Use the **include(...)** method to specify the types and subtypes required.

E.g. to process only type 30 records:

```
try (SmfRecordReader reader =
          SmfRecordReader.fromName(inputFile)
              .include(30))
{
    // Process type 30 records here
}
```

To process only type 30 subtype 5 records:

```
try (SmfRecordReader reader =
          SmfRecordReader.fromName(inputFile)
              .include(30, 5))
{
    // Process type 30 subtype 5 records here
}
```

The include() method returns the SmfRecordReader so it can be chained as many times as required. To process only type 30 subtypes 2 and 3 records:

```
try (SmfRecordReader reader =
          SmfRecordReader.fromName(inputFile)
              .include(30, 2)
              .include(30, 3))
{
    // Process type 30 subtypes 2 and 3 records here
}
```

If no include(...) is specified, all record types are included.

EasySMF:JE Samples

Sample reports, scripts and JCL are available on Github:

https://github.com/BlackHillSoftware/easysmf-samples

Samples include:

- A tutorial to introduce and demonstrate various EasySMF:JE concepts
- Programs to report and remove duplicate SMF data
- Programs to convert various SMF records to JSON format
- Skeleton projects which you can copy to create your own EasySMF:JE projects
- JCL to run on z/OS.

A copy of the samples from Github is included in the distributed pax and zip files.

Working with SMF records

The SMFRecordReader implements the **Iterable<SmfRecord>** interface.

This means that it can be the target of a for-each loop operating on SmfRecords. To read SMF records from the reader:

```
try (SmfRecordReader reader =
        SmfRecordReader.fromName(args[0]))
{
    for (SmfRecord record : reader)
    {
        // Process each record here
    }
}
```

The SmfRecord class is the base class for all SMF record types. It implements a number of methods common to all SMF record types, e.g. recordType().

Typically you will need to transform the SmfRecord into a specialized record type, e.g. Smf30Record to access the data. The record classes have a static **from** method to create a the specialized record from another record.

Record Sections

After creating the specialized record you can use the record methods to get the sections and data fields.

Single Sections vs. List<>

Some sections e.g the SMF 30 IdentificationSection are always present and there are never more than one in a record. These sections are accessed using a method that returns a single section.

Other sections e.g. SMF 30 ExcpSection can occur multiple times in a record. These sections are accessed using a method returning a list of 0 or more sections, depending how many were present in the record.

There are some sections that never occur more than once, but are not always present. These can be accessed using a method returning a list of 0 or 1 sections, or using a method returning the single section or null.

The reason both methods exist is that these sections are located in the record by SMF "triplets" with a count field specifying how many sections are present. The List<> method is provided for consistency with other sections using triplets. In cases where it is documented that there can only be 1 or 0 sections, a method is also provided to return the single section.

Which you choose depends on personal preference.

E.g. the Smf30Record.processorAccountingSections() method returns a list containing either 1 or 0 ProcessorAccountingSections depending on whether the section is present in the record:

```
for (SmfRecord record : reader)
```

```
    {

        Smf30Record r30 = Smf30Record.from(record);

        for (ProcessorAccountingSection procAcct
                : r30.processorAccountingSections())
        {

            Duration cpuTime = procAcct.smf30cpt()
                    .plus(procAcct.smf30cps());
            if (cpuTime.getSeconds() >= 60)

            {

                System.out.format("%-23s %-8s %12s%n",

                   r30.smfDateTime(),
                   r30.identificationSection().smf30jbn(),
                   cpuTime);
            }
        }

    }
```

In this case there is also a processorAccountingSection() method which returns the ProcessorAccountingSection if present, or null. An alternate way to process this particular section is:

```
    for (SmfRecord record : reader)

    {

        Smf30Record r30 = Smf30Record.from(record);

        ProcessorAccountingSection procAcct = r30.processorAccountingSection();
        if (procAcct != null)
        {
            ...
        }
    }
```

Data Fields

Records and Sections provide methods to access their data fields.

E.g. the SMF 30 IoActivitySection provides smf30tex() to get the value of the SMF30TEX field with the job or step EXCP count.

Data Types

SMF data types are converted to Java data types as follows:

- **1, 2 and 3 byte integer** values are converted to the Java **int** datatype (32 bits). Java integers are signed, so 32 bits is too small for a 4 byte/32 bit unsigned value.
- **4 byte/32 bit unsigned integer values** are converted to a Java **long** (64 bits).
- **8 byte/64 bit unsigned integer** values are available as **long** and **BigInteger** values. The long data type may provide better performance if the value will not exceed the maximum 64 bit signed value. The API will throw an exception if the value is too large for a long. If this is possible, use the BigInteger value.
- **Integers greater than 8 bytes** are converted to **BigInteger**s.
- **Floating point** values are converted to Java **double**s.

Dates and Times

EasySMF:JE converts the various SMF dates and times to **java.time** classes. This allows you to use date and time values without doing your own conversions or knowing the units of each field.

Java.time has has nanosecond precision, which is enough to represent most SMF fields. The raw unconverted SMF values are also available for all time based fields if required.

The java.time classes also provide facilities to determine the day of the week, timezone rules for daylight saving etc.

Different types of date and time are represented by different classes. This avoids errors that can occur from using incompatible types, e.g. comparing local times and GMT/UTC times. To do these comparisons you need to explicitly convert between types, for example you can convert from a LocalDate to a LocalDateTime using the LocalDate.atTime(LocalTime) method.

Likewise, you can convert from LocalDateTimes to ZonedDateTimes by applying a ZoneOffset or ZoneId. If you have systems in different time zones you could apply the appropriate ZoneId for each system to synchronize SMF reports, or even report using a different time zone.

Common java.time classes used in EasySMF:JE are:

- **Duration** - A length of time, e.g. CPU time or connect time for a job.
- **LocalDate** - A date without a time and with unknown timezone.
- **LocalTime** - A time of day, without any date information.
- **LocalDateTime** - Incudes both time and date, with unknown time zone.
- **ZonedDateTime** - Includes date, time and time zone information. Typically in EasySMF these are from SMF fields expressed in GMT time. If your clock is not set to GMT these will be incorrect.
- **ZoneOffset** - An offset from GMT time. The system time zone information is stored in some records.

Durations are also converted to double precision floating point values in seconds to simplify calculations.

SMF Record Notes

CICS

CICS Record Compression

CICS can compress SMF records as they are created using CSRCESRV data compression. Compressed CICS records will be decompressed when the Smf110Record is constructed.

CICS Monitoring Facility Records

Accessing data from CICS monitoring facility records is slightly different to other SMF records because the data needs to be accessed using a Dictionary.

Dictionary records are handled automatically, however you cannot access the data from a record before a related dictionary record has been seen. You can check whether a dictionary record is available using Smf110Record.haveDictionary() or simply concatenate all required dictionary records ahead of the data records in the input data.

Specific fields are defined by name and type using entries from com.blackhillsoftware.smf.cics.monitoring. Then Performance records are read from the SMF record, and specific fields accessed using getField(...) methods or variations.

```
try (SmfRecordReader reader =
         SmfRecordReader
             .fromDD("INPUT")
             .include(110, Smf110Record.SMFMNSTY))     // include only type 110 subtype 1 recor
 {
     for (SmfRecord record : reader)                   // read each record
     {
         Smf100Record r110 = Smf110Record.from(record);
         if (r110.haveDictionary())
         {
             for (PerformanceRecord perfdata :
                 r110.performanceRecords())            // process 0 or more PerformanceRecord s
             {
                 String txName = perfdata.getField(Field.TRAN);
                 ZonedDateTime start = perfdata.getField(Field.START);
                 ZonedDateTime stop = perfdata.getField(Field.STOP);
                 double dispatch = perfdata.getFieldTimerSeconds(Field.USRDISPT);

                 //...  process data
             }
         }
     }
 }
```

CICS Fields are defined in the **com.blackhillsoftware.smf.cics.monitoring.fields.Field** class so their nickname can be used to access the data e.g.
`getField(Field.TRAN)`
to get the transaction name.

Fields can also be accessed by ID, by explicitly defining your own Field:

```
        ByteStringField transaction = ByteStringField.define("DFHTASK", "C001");
```

```
String txName = perfdata.getField(transaction);
```

Fields with a simple value e.g. type C byte string fields are accessed using the getField(...) methods.

CICS Clock fields contain multiple values so they also use multiple getField methods.

- **getFieldTimer(...)** and **getFieldTimerSeconds(...)** get the timer as a **Duration** and **double** value in seconds respectively.
- **getFieldFlags(...)** gets the CICS clock flag bits.
- **getFieldPeriodCount(...)** gets the period count value from the CICS clock.
- **getField(...)** gets the entire CicsClock. You can then retrieve the components as required.

SLF4J Message Logging

Messages from EasySMF are written using Simple Logging Facade for Java (SLF4J).

SLF4J is a message logging framework for Java. It allows messages to be directed to various locations without changing the source program. EasySMF:JE uses SLF4J to write various status messages.

The **slf4j-simple** binding is included which directs EasySMF messages to standard error (stderr). Other bindings are available to direct messages to other destinations, e.g. if the program stderr is not available or not monitored. You may already have other Java programs using SLF4J with your own logging standards.

See http://www.slf4j.org/ to see what options are available.

Appendices

Appendix A - 3rd Party Licenses

The EasySMF:JE distribution includes some 3rd party software. License agreements are included here where required.

SLF4J

Simple Logging Facade for Java (SLF4J)

Gson

Apache Commons CLI

Index