Anyframe Logback Plugin

Version 1.1.0

I.Introduction

Logback plugin은 Logback framework에서 제공하는 SiftingAppender와 MDC를 활용한 사용자별 로깅 방법을 가이드 하기 위한 샘플코드, 참조 라이브러리, 설정파일로 구성되어 있다.

Installation

Command 창에서 다음과 같이 명령어를 입력하여 Logback plugin을 설치한다.

```
mvn anyframe:install -Dname=logback
```

installed(mvn anyframe:installed) 혹은 jetty:run(mvn clean jetty:run) command를 이용하여 설치 결과를 확인해볼 수 있다.

Plugin Name	Version Range
core [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/htmlsingle/core.html]	2.0.0 > * > 1.4.0

주의 사항

- Logback Plugin을 설치한 후 샘플 어플리케이션 실행 시 slf4j-log4j-x.jar 라이브러리와 logback-classic-x.jar 라이브러리가 충돌하여 "SLF4J: Class path contains multiple SLF4J bindings" 라는 경고 메시지가 나타날 수 있으나 어플리케이션 실행에는 영향을 미치지 않는다. 해당 경고 메시지를 제거하기 위해서는 slf4j-log4j 라이브러리를 수동으로 제거하면 문제를 해결할 수 있다.
- Logback을 사용하는 경우 Core Plugin을 제외한 다른 Plugin 설치시 log4j.xml에 자동으로 추가되는 logger는 직접 logback.xml에 등록해주어야한다.
- 기본적으로 Anyframe Plugin 설치로 생긴 각각의 샘플 코드는 LoggingAspect.java 클래스에 구현된 로직을 바탕으로 로그를 출력하고 있다. 이 때, LoggingAspect.java 클래스에 의해 출력되는 로그는 Debug 레벨로 설정되어 있으므로 Logback.xml에 정의된 해당 Logger의 Level을 Debug로 변경하여야 한다.
- Logback Plugin 설치 시 등록되는 FileAppender와 RollingFileAppender는 로그 파일이 생성되는 위치를 상대경로로 설정되어있다. 상대경로의 경우 해당 로그 파일은 샘플 어플리케이션을 실행한 곳을 기준으로 생성이 된다. 예를 들면 eclipse 와 eclipse에 연결 된 tomcat을 이용하여 샘플 어플리케이션을 실행한 경우에는 eclipse가 설치된 경로이고 특정 WAS에 배포하여 사용하는 경우에는 해당 WAS에 실행되는 상대경로를 확인하여야 한다.

II.Logback

Logback은 널리 알려진 Java Logging Framework인 Log4j를 대체하기 위해 개발된 Logging Framework이다. Logback은 log4j에 비해서 속도나 메모리 사용면에서 개선되었고, slf4j 인터페이스를 직접 구현함으로써 로깅을 수행하는 어플리케이션에서는 slf4j api만을 사용하여 로깅을 수행할 수 있기때문에 다른 Logging framework 와의 변경을 손쉽게 할 수 있다. 또한 Logback은 Filter, SiftingAppender, Dynamic Reloading, 설정파일의 조건부 처리 기능, RollingFileAppender의 자동압축기능과 설정한 기간이 지난 로그 파일의 자동 삭제와 같은 새로운 기능을 지원한다.

이 장에서는 Logback의 구성요소, 설정방법, 기능에 대해서 알아보도록 한다.

Dependencies

Logback을 사용하기 위해서는 logback-core와 logback-classic 라이브러리가 필수적으로 필요하다. 또한 Logback은 sf4j를 구현하였기 때문에 slf4j-api 라이브러리를 필요로 한다. 다음은 필수라이브러리에 대한 추가 설명이다.

- slf4j-api : slf4j 표준 인터페이스 구현체로, core Plugin 설치시 디폴트로 추가된다.
- logback-core : logback-classic과 의존 모듈로 appender, layout 등이 구현되어 있는 모듈이다.
- logback-classic : Logback의 logger와 SiftAppender등이 구현되어 있는 모듈이다.

Logback 에서는 <if> 태그를 제공하여 하나의 설정파일을 다양한 조건에 의해 변경할 수 있는 조건부 처리 기능을 제공하는데 이 조건을 처리하는 방식을 Java언어를 이용한 방식과 groovy언어를 사용한 방식 두 가지를 지원한다. Logback Plugin에서는 Java를 이용한 방식을 사용하였고 설정파일에서 JAVA 표현식을 사용하기 위해 필요한 라이브러리인 janino와 commons-complier 라이브러리가 추가된다.

core 모듈이 설치되면 sl4j-log4j-x.jar가 참조 라이브러리로 지정이 된다. sl4j-log4j-x.jar와 logback-classic-x.jar 이 동시에 존재하게 되면, slf4j-api 구현체가 중복으로 존재하게 되므로, Logback Plugin 설치시 slf4j-log4j-x.jar에 대한 dependency를 제거 한다.

1.Configuration

Logback은 구성요소 측면에서 Log4j와 유사한 구조를 가지고 있다. Logback을 이용하여 로깅을 수행하기 위해서 필요한 주요 설정요소로는 Logger, Appender, Encoder의 3가지가 있으며 각각은 다음과 같은 역할을 수행한다.

- Logger : 실제 로깅을 수행하는 구성요소로 Level 속성을 통해서 출력할 로그의 레벨을 조절할 수 있다.
- Appender : 로그 메시지가 출력될 대상을 결정하는 요소
- Encoder : Appender에 포함되어 사용자가 지정한 형식으로 표현 될 로그메시지를 변환하는 역할을 담당하는 요소

Logback에서 설정파일을 작성하는 방법은 크게 두 가지가 있다.

- XML을 이용한 설정방법 : logback.xml로 설정 파일 작성 후 해당 파일을 클래스패스에 위치시킨다.
- Groovy 언어를 이용한 설정방법 : logback.groovy로 설정 파일 작성 후 해당 파일을 클래스패스에 위치시킨다.

이 매뉴얼에서는 위에서 언급한 Logback 설정요소들과 xml을 이용한 설정 방법 에 대해서 설명하고자 한다. Groovy언어를 이용한 설정방법에 대해서는 여기 [http://logback.qos.ch/manual/groovy.html]를 참조하도록 한다.

1.1.Appender

Appender는 로그를 수행할때 로그 정보를 어디에 기록할지를 결정한다. 로그정보를 기록할 수 있는 곳으로는 콘솔, 파일, DB, 메일, JMS등이 있다. 다음은 logback에서 제공하는 Appender와 설정 방법을 기술하였다.

1.1.1.ConsoleAppender

ConsoleAppender는 로그정보를 콘솔로 출력하는 기능을 수행한다. 다음은 ConsoleAppender를 설정하는 예제이다.

Pattern 속성을 지정하여 어플리케이션이 시작한 이후로 부터 로그이벤트가 생성된 상대적 시간(ms), 쓰레드명, 로그레벨, 로거명, 로그 메시지를 출력하도록 설정한 예제이다. encoder에 대한 좀 더 자세한 설명은 encoder를 참조한다.

```
<appender name="console" class="ch.qos.logback.ConsoleAppender">
    <encoder>
        <pattern>%-4relative [%thread] %-5level %logger{35} - %msg %n</pattern>
    </encoder>
</appender>
```

다음은 위에서 정의한 ConsoleAppender를 활용하여 출력한 로그메시지의 일부이다.

```
215  [main] INFO  consoleLogger - console Appender Test
220  [main] DEBUG consoleLogger - this is a debug log message
```

1.1.2.FileAppender

FileAppender는 로그정보를 특정 파일로 출력하는 기능을 수행한다.

FileAppender의 주요속성들은 다음과 같다.

속성명	기능	타입	기본값
append	기존에 존재하는 파일에 로그를 남길때 해당 파일에 추가하여 로그를 남길지 여부를 설정하는 속성	boolean	true
file	로그를 출력할 파일명을 설정할 수 있는 속성. 해당 경로에 파일이 존재하지 않을 경우 해당 파일 생성	String	N/A
prudent	다수의 JVM에 하나의 파일에 로그를 안전하게 남길수 있도록 동기화를 할지 여부를 결정하는 속성	boolean	false

다음은 FileAppender를 설정하는 예제이다.

```
<appender name="file" class="ch.qos.logback.core.FileAppender">
    <file>testFile.log</file>
    <encoder>
        <pattern>%-4relative [%thread] %-5level %logger{35} - %msg %n</pattern>
    </encoder>
</appender>
```

1.1.3.RollingFileAppender

하나의 파일에 모든 로그를 남기는 방법은 로그의 검색, 관리적인 면에서 효율적이지 못하다. RollingFileAppender는 사용자가 설정한 조건 (파일크기, 시간 등)에 따라서 로그파일을 변경하여 로그정보를 출력하는 기능을 수행한다.

RollingFileAppender은 기본적으로 FileAppender의 모든 속성들을 가지고 있고 RollingAppender의 추가적인 속성들은 다음과 같다.

속성명	기능	타입	기본값
triggeringPolicy	어떤 조건에서 rolling 이벤트가 발생할 지 조건을 결정하는 속성	ch.qos.logback.core.rolling.TriggeringPolicy	N/A
rollingPolicy	rolling이 발생할 때 어떤 작업을 수행할지 결정하는 속성	ch.qos.logback.core.rolling.RollingPolicy	N/A

Logback에서는 다양한 TriggerPolicy와 RollingPolicy 구현체를 제공하고 있다. 다음은 Logback에서 제공하는 주요 Policy에 대한 설명과 그 설정 방법을 기술하였다.

다음은 TimeBasedRollingPolicy를 이용하여 날짜별 로깅을 수행하는 예제이다. <file> 태그를 통해 현재 로깅이 수행되는 파일명을 지정할 수 있고, 로깅이 완료된 파일은 fileNamePattern에 패턴에 맞춰서 파일명이 변경된다. TimeBasedRollingPolicy의 fileNamePattern에 속성에 설정된 날짜패턴인 %d{yyyy-MM-dd}에 따라 일별로 로그를 남기게 하였고, maxHistory 속성을 설정하여 최근 30개의 로그정보만 남기고 그 이후의 파일은 자동으로 삭제하도록 설정한 예제이다. fileNamePattern속성에 설정 가능한 날짜패턴에 대한 상세한 설명은 여기 [http://logback.qos.ch/manual/appenders.html#TimeBasedRollingPolicy]를 참조한다.

```
<appender name="dailyRolling" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>currentLog.log</file>

     <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
         <fileNamePattern>logFile.%d{yyyy-MM-dd}.log</fileNamePattern>
         <maxHistory>30</maxHistory>
    </rollingPolicy>

    <encoder>
        <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n</pattern>
    </encoder>
```

```
    </appender>
```

다음은 SizeBasedTriggeringPolicy를 이용하여 5MB 파일크기로 Rolling을 수행하고 FixedWindowRollingPolicy를 이용하여 최근 3개의 로그 파일만 유지하도록 설정한 예제이다.

```
    <appender name="fileSizeRolling"
 class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>currentLog.log</file>

        <rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
            <fileNamePattern>tests.%i.log.zip</fileNamePattern>
            <minIndex>1</minIndex>
            <maxIndex>3</maxIndex>
        </rollingPolicy>

        <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
            <maxFileSize>5MB</maxFileSize>
        </triggeringPolicy>

        <encoder>
            <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n</pattern>
        </encoder>
    </appender>
```

1.1.4.AsyncAppender

AsyncAppender는 logback 1.0.4 버전에 새롭게 추가된 appender로 기존의 appender가 로그를 출력할 때 비동기적으로 처리할 수 있는 기능을 추가한다.

AsyncAppender의 주요 속성은 다음과 같다.

속성명	기능	타입	기본값
queueSize	AsyncAppender 로그이벤트를 저장하는 용도로 사용하는 Queue의 크기를 설정하는 속성	int	256
discardingThreshold	Queue의 크기중에 비어있는 부분이 discardingThreshold 비율 이하인 경우 INFO 레벨 이하의 로그는 저장하지 않는다. 해당 경계비율을 설정하는 속성 따라서 모든 로그 이벤트를 처리하기 위해서는 해당 속성값을 0으로 설정해야 한다.	int	20
includeCallerData	해당 로그 이벤트 정보를 Queue에 추가하는 시점에 로그를 호출한 정보 (callerData)를 추출할지 여부를 결정하는 속성	boolean	false

다음은 AsyncAppender의 설정 예제이다. AsyncAppender에 실제 로그를 수행할 appender를 appender-ref 속성으로 설정하고, 해당 기능을 사용하기 위한 Logger 에서 AsyncAppender를 참조하도록 설정해야한다.

```
    <appender name="file" class="ch.qos.logback.core.FileAppender">
        <file>testFile.log</file>
        <encoder>
            <pattern>%-4relative [%thread] %-5level %logger{35} - %msg %n</pattern>
        </encoder>
    </appender>

    <appender name="asyncFile" class="ch.qos.logback.classic.AsyncAppender">
        <appender-ref ref="file"/>
    </appender>
```

```
<logger name="asyncFileLogger" level="INFO" additivity="false">
    <appender-ref ref="asyncFile" />
</logger>
```

1.1.5.DBAppender

DBAppender는 로그정보를 DB에 insert하는 역할을 한다. Log4j의 JDBCAppender와는 달리 Logback의 DBAppender는 디폴트로 LOGGING_EVENT, LOGGING_EVENT_PROPERTY, LOGGING_EVENT_EXCEPTION의 3가지 테이블을 기본적으로 필요로 한다. 다음은 각 테이블과 컬럼에 대한 설명이다. 타입은 실제 사용하는 DBMS 환경에 따라 변경될 수 있다.

1.1.5.1.LOGGING_EVENT 테이블

LOGGING_EVENT 테이블은 로그메시지, timestamp 등 일반적인 로그 이벤트 정보를 저장하는 역할을 하는 테이블이다. 테이블의 각 컬럼은 다음과 같다.

컬럼명	타입	기능
timestmp	big int	로그 이벤트 발생시점에 대한 타임스탬프 값
formatted_message	text	로깅시점에 전달된 메시지와 파라미터를 이용하여 변환된 메시지
logger_name	varchar	로그를 출력할 당시 사용된 Logger의 name
level_string	varchar	로그를 출력할 당시 level
reference_flag	smallint	reference_flag는 로그 이벤트발생 당시 MDC 혹은 Context 에 저장된 속성이 있어서 LOGGING_EVENT_PROPERTY에 데이터가 저장되는 경우 1의 값을 가지고 로그 이벤트에 Exception 객체에 대한 정보가 있어 LOGGING_EXCEPTION에 저정되어야 할 데이터가 있는 경우 2의 값을 가진다. 두가지 데이터가 모두 있는 경우 3의 값을 가진다.
caller_filename	varchar	로그를 호출한 파일명
caller_class	varchar	로그를 호출한 클래스명
caller_method	varchar	로그를 호출한 메소드명
caller_line	char	로그를 호출한 라인번호
event_id	int	로그 이벤트에 따른 DB의 고유한 키 값

1.1.5.2.LOGGING_EVENT_PROPERTY 테이블

LOGGING_EVENT_PROPERTY 테이블은 로깅 이벤트가 발생했을 때 컨텍스트에 저장된 프로퍼티 정보와 MDC에 저장된 정보를 저장하는 용도로 사용되는 테이블이다. 테이블의 각 컬럼은 다음과 같다.

컬럼명	타입	기능
event_id	int	로그 이벤트에 따른 DB의 고유한 키 값
mapped_key	varchar	MDC 혹은 context에 저장된 키 값
mapped_value	text	MDC 혹은 context에 저장된 키에 해당하는 value

1.1.5.3.LOGGING_EXCEPTION 테이블

LOGGING_EXCEPTION 테이블은 로깅이벤트가 발생했을 때, 로그메시지와 함께 파라미터로 Exception 객체를 넘겨주는 경우 Exception 정보를 이용하여 stackTrace 정보를 저장하는 테이블이다. 테이블의 각 컬럼은 다음과 같다.

컬럼명	타입	기능
event_id	int	로그 이벤트에 따른 DB의 고유한 키 값
i	smallint	stack trace에 대한 인덱스 정보
trace_line	varchar	해당 stack trace에 대한 메시지

DBAppender의 주요 속성은 다음과 같다.

속성명	기능	타입	기본값
connectionSource	DBAppender에서 connection 정보를 얻기 위해서 사용하는 클래스를 지정하는 속성으로 Logback에서는 DriverManagerConnectionSource, connection pool 기능을 이용할 수 있는 DataSourceConnectionSource, JNDI로 부터 DataSource를 이용할 수 있는 JNDIConnectionSource를 지원한다.	ch.qos.logback.core.db.ConnectionSource	N/A

다음은 Apache Commons DBCP를 이용하여 DBAppender를 설정한 예제이다. db접속정보는 logback-db.properties 파일에 두고, 해당 프로퍼티 파일에 설정된 값을 불러와서 사용하도록 하였다. 프로퍼티 파일을 사용하기 위해서는 <property>태그의 file 속성에 프로퍼티 파일의 경로를 명시해 주면된다. 프로퍼티 파일의 경로를 상대경로를 사용한 경우 프로젝트 루트를 기준으로 한 경로를 명시해주면 된다.

```
<property file="src/test/resources/logback-db.properties" />

<appender name="dataSourceDB" class="ch.qos.logback.classic.db.DBAppender">

     <connectionSource class="ch.qos.logback.core.db.DataSourceConnectionSource">
        <dataSource class="org.apache.commons.dbcp.BasicDataSource">
            <driverClassName>${driver}</driverClassName>
            <url>${url}</url>
            <username>${username}</username>
            <password>${password}</password>
        </dataSource>
    </connectionSource>
</appender>
```

DBAppender는 기본으로 동기적으로 로그를 수행한다. DBAppender와 AsyncAppender를 함께 사용하면 비동기적으로 DB에 로그를 남길 수 있다.

```
<appender name="db" class="ch.qos.logback.classic.db.DBAppender">
     ....
</appender>

<appender name="asyncDB" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="db"/>
    <includeCallerData>true</includeCallerData>
</appender>

<logger name="asyncDBLogger" level="DEBUG" additivity="false">
    <appender-ref ref="asyncDB"/>
</logger>
```

주의 사항

DBAppender와 AsyncAppender를 활용하여 비동기 로깅을 할때 AsyncAppender의 includeCallerData속성을 반드시 true로 설정해야 한다.

1.1.5.4.사용자 정의 컬럼명 사용

Logback에서 제공하는 DBAppender의 경우 미리 정의된 테이블과 컬럼구조를 사용하고 있기 때문에, 사용자 정의 테이블 혹은 sql을 사용하기 위해서는 관련 클래스의 확장이 필요하다. DBAppender에서 사용하는 테이블의 개수와 구조를 동일하게 유지하면서 단순히 컬럼이나 테이블명만을 변경하려고 하는 경우는 DBAppender에서 내부적으로 사용하는 클래스인 DBNameResolver를 새로 구현하는 방법을 고려해 볼 수 있다. Logback에서 제공하는 DBNameResolver 인터페이스를 구현하는 클래스를 개발하고 DB Appender에서 해당 DBNameResolver를 사용하도록 설정할 수 있다.

다음은 MyDBNameResolver라는 사용자 정의 DBNameResolver를 구현하고 설정하는 예제이다.

```
public class MyDBNameResolver  implements DBNameResolver{

    private static Map<String, String> columnNameMap = new HashMap<String, String>();
    private static Map<String, String> tableNameMap = new HashMap<String, String>();

    static{
         tableNameMap.put("LOGGING_EVENT", "MY_LOGGING_EVENT");
         tableNameMap.put("LOGGING_EVENT_PROPERTY", "MY_LOGGING_EVENT_PROPERTY");
         tableNameMap.put("LOGGING_EVENT_EXCEPTION", "MY_LOGGING_EVENT_EXCEPTION");
    }

    public <N extends Enum<?>> String getColumnName(N columnName) {
        return columnNameMap.get(columnName.toString());
    }

    public <N extends Enum<?>> String getTableName(N tableName) {
        return tableNameMap.get(tableName.toString());
    }
 ......
```

주의 사항

logback 1.0.7 버전까지 사용자 정의 DBNameResolver를 추가하여 컬럼명을 변경할 경우에도 EVENT_ID 라는 컬럼을 변경하면 제대로 동작하지 않을 수 있으므로 주의해야 한다.

다음은 사용자 정의 DBNameResolver를 DBAppender에 설정하는 예제이다.

```
<appender name="customDB" class="ch.qos.logback.classic.db.DBAppender">
    <connectionSource class="ch.qos.logback.core.db.DataSourceConnectionSource">
        <dataSource class="org.apache.commons.dbcp.BasicDataSource">
            <driverClass>${driver}</driverClass>
            <url>${url}</url>
            <user>${username}</user>
            <password>${password}</password>
        </dataSource>
    </connectionSource>

    <dbNameResolver class="org.anyframe.logback.appender.db.MyDBNameResolver"/>

</appender>
```

1.1.6.SocketAppender

SocketAppender는 로컬에서 발생한 로그정보를 TCP Socket을 이용하여 remote에 있는 서버로 전송하는 역할을 한다. remoteHost, port 속성을 이용하여 서버의 IP, port를 설정할 수 있다. SocketAppender의 includeCallerData 속성을 true로 하면 클라이언트의 callerData 정보를 포함하여 서버로 전송한다. 다음은 SocketAppender를 설정하는 예제이다.

```
<appender name="socket" class="ch.qos.logback.classic.net.SocketAppender">
    <includeCallerData>true</includeCallerData>
    <port>${serverPort}</port>
    <remoteHost>${remoteHost}</remoteHost>
</appender>
```

SocketAppender를 사용하는 경우 주의할 점은 SocketAppener의 경우 단순히 LoggingEvent 정보를 직렬화 하여 서버로 전송하는 역할만 수행하므로 layout 혹은 encoder를 설정하지 않아야 한다는 점이다.

SocketAppender를 테스트 하기 위해서는 서버가 필요하다. 다음은 SocketAppender를 이용하여 logging 테스트를 수행하기 위한 샘플 서버인 SimpleSocketServer클래스와 SocketNode 클래스의 일부분이다. SimpleSocketServer 클래스는 전체 클라이언트와 연결과 SocketNode들을 관리하는 역할을 하고 SocketNode 클래스에서는 하나의 클라이언트가 로그 요청을 전송할때 마다 해당정보를 읽어서 로그를 출력하는 역할을 담당한다. 전체 샘플 서버 코드를 확인하기 위해서는 여기를 참조한다.

```
public class SimpleSocketServer extends Thread {
    private ServerSocket serverSocket;
    private List<SocketNode> socketNodeList = new ArrayList<SocketNode>();
    private boolean closed = false;

    public void run() {
        try {
        serverSocket = new ServerSocket(port);
        while (!closed) {
            ...
            Socket socket = serverSocket.accept();
            SocketNode newSocketNode = new SocketNode(this, socket, lc);
            synchronized (socketNodeList) {
                socketNodeList.add(newSocketNode);
            }
            new Thread(newSocketNode).start();
        }
    } catch (Exception e) {
        if (closed) {
            logger.info("Exception in run method for a closed server. This is normal.");
        } else {
            logger.error("Unexpected failure in run method", e);
        }
    }
...
 }
```

```
public class SocketNode implements Runnable {

    public void run() {
        ILoggingEvent event;
        Logger remoteLogger;

        try {
            while (!closed) {

               event = (ILoggingEvent) ois.readObject();
            remoteLogger = context.getLogger(event.getLoggerName());
            if (remoteLogger.isEnabledFor(event.getLevel())) {
                remoteLogger.callAppenders(event);
            }
               ......
            }
     }
```

1.1.7.SMTPAppender

SMTPAppender는 로그요청을 버퍼에 저장하고 특정 이벤트가 발생하면 저장된 로그를 메일로 전송하는 역할을 한다. 메일 전송 시 기본적으로 비동기 방식으로 전송하게 된다. 이때 Logback의 SMTPAppender는 버퍼에 저장된 만큼 메일을 전송하는 것이 아니고, 버퍼에 저장된 로그내용을 하나의 본문으로 합쳐서 한건의 메일로 발송하는 특징을 가지고 있다. SMTPAppender는 Java Mail 라이브러리를 이용하여 메일을 전송하므로 SMTPAppender를 사용하기 위해서는 java mail 라이브러리(mail-x.x.jar)와 activation(activation-x.x.jar) 라이브러리가 필요하다.

SMTPAppender의 주요속성은 다음과 같다.

속성명	기능	타입	기본값
smtpHost	SMTP 서버의 호스트 명	String	N/A
smtpPort	SMTP 서버의 포트	int	25
to	메일 수신자 패턴. 한번에 여러명의 수신자를 지정하기 위해서는 콤마로 분리하거나 혹은 to태그를 여러번 반복사용하여 지정가능하다.	String	
from	메일 송신자	String	N/A
subject	메일 전송시 지정할 메일의 제목 패턴. PatternLayout의 형식을 사용할 수 있다.	String	%logger{20} - %m
discriminator	로그정보를 어떤 버퍼에 저장할지 결정하는 속성. SMTPAppender에서는 discriminator가 반환하는 키 값에 따라서 로그이벤트 정보를 서로 다른 버퍼에 저장한다. 기본적으로 DefaultDiscriminator가 설정되어 있고 이 객체는 항상 같은 키 값을 반환하므로 하나의 버퍼에 로그이벤트 정보를 저장한다.	ch.qos.logback.core.sift.Discriminator	ch.qos.logback.core.sift.Defau
evaluator	버퍼에 저장된 로그정보를 이용하여 어느 시점 메일을 발송할지 여부를 결정하는 속성. SMTPAppender 에서는 evaluator가 true를 리턴하는 경우 해당 버퍼에 저장된 정보를 이용하여 메일을 발송한다. 기본적으로 OnErrorEvaluator가 설정되어 있고 이 객체는 에러로그 발생시 true를 리턴하므로 에러로그 발생시 메일이 발송이 된다.	ch.qos.logback.core.boolex.EvaluationEvaluator	ch.qos.logback.core.boolex.
username	메일서버 인증을 위한 사용자명	String	null
password	메일서버 인증을 위한 비밀번호	String	null
STARTTLS	인증시에 STARTTLS 프로토콜을 사용할지를 설정하는 여부	boolean	false
SSL	인증시에 SSL 프로토콜을 사용할지를 설정하는 여부	boolean	false
asynchronousSending	비동기 메일전송 여부	boolean	true
charsetEncoding	메일 전송시 인코딩	String	UTF-8
sessionViaJNDI	메일 사용할 Session객체를 JNDI로부터 얻어오게 할지 여부	boolean	false
jndiLocation	JNDI로 객체를 Session을 얻어올때 사용할 location명	String	java:comp/env/mail/Session

다음은 SMTPAppender를 사용한 기본적인 예제이다. SSL속성을 true로 하여 인증시 SSL 프로토콜이 적용되게 설정한 예제이다.

```
<appender name="email" class="ch.qos.logback.classic.net.SMTPAppender">
    <smtpHost>${smtpHost}</smtpHost>
    <smtpPort>${smtpPort}</smtpPort>
    <SSL>true</SSL>
    <username>${userName}</username>
    <password>${password}</password>
    <to>${to}</to> <!-- additional destinations are possible -->
    <from>${from}</from>

    <subject>TESTING-Email: %logger{20} - %m</subject>
    <layout class="ch.qos.logback.classic.PatternLayout">
        <pattern>%date %-5level %logger{35} - %message%n</pattern>
    </layout>
</appender>
```

SMTPAppender는 로그 요청이 올때마다 매번 메일을 전송하는 것에 대한 부하가 크기 때문에 기본적으로 버퍼에 저장을 하고 특정 조건이 되면 버퍼에 저장된 내용을 발송한다. 이렇게 로그 요청을 이용하여 단순히 버퍼에 저장할지 메일을 발송할지 여부를 결정하는 속성이 EventEvaluator이다. 기본적으로는 OnErrorEvaluator가 설정되어 있고, OnErrorEvaluator는 에러 이상의 레벨의 로그요청이 오는 경우 메일을 보낸다. 이 Evaluator 속성을 변경하면 필요한 조건에 메일이 발송되도록 설정할 수 있다.

다음은 Logback에서 제공하는 OnMarkerEvaluator를 이용하여 로그 요청에 특정 Marker의 값이 들어오는 경우에 메일을 전송하도록 설정한 예제이다.

```
<appender name="EMAIL_MARKER_EVALUATOR" class="ch.qos.logback.classic.net.SMTPAppender">
    <smtpHost>${smtpHost}</smtpHost>
    <smtpPort>${smtpPort}</smtpPort>
    <SSL>true</SSL>
    <username>${userName}</username>
    <password>${password}</password>
    <to>${to}</to> <!-- additional destinations are possible -->

    <from>${from}</from>

    <subject>TESTING-ASYNC: %logger{20} - %m</subject>
    <layout class="ch.qos.logback.classic.PatternLayout">
        <pattern>%date %-5level %logger{35} - %message%n</pattern>
    </layout>

    <evaluator class="ch.qos.logback.classic.boolex.OnMarkerEvaluator">
        <marker>NOTIFY_ADMIN</marker>
        <marker>TRANSACTION_FAILURE</marker>
    </evaluator>

</appender>
```

해당 설정파일을 이용하여 Marker를 이용한 로그를 남기는 예제는 다음과 같다.

```
logger.warn("the first buffered message");
// even the error message will be buffered
logger.error("the buffered error log message");
Marker notifyAdmin = MarkerFactory.getMarker("NOTIFY_ADMIN");
logger.error(notifyAdmin, "NOTIFY_ADMIN");
```

SMTPAppender는 기본적으로 하나의 버퍼를 사용하여 로그이벤트를 처리한다. Discriminator를 이용하면 특정조건에 따라 로그이벤트를 여러개의 버퍼에 저장하게 되고, 따라서 기준에 따라 분류된 로그내용을 단위로 메일로 전송하는 것이 가능하다. Logback에서는 MDCBasedDiscriminator를 제공하고 이 클래스는 로그이벤트 처리 시 MDC에 저장된 값에 따라서 서로 다른 버퍼를 사용하게 된다.

다음은 MDCDiscriminator를 이용한 설정 예제이다. MDC에 있는 req.remoteHost 키 값을 이용하여 IP 별로 로그 이벤트가 서로 다른 버퍼에 저장되도록 설정하였고, 메일 제목에 MDC에 저장된 값을 출력해주도록 하였다.

```
   <appender name="EMAIL_MDC_DISCRIMINATOR"
class="ch.qos.logback.classic.net.SMTPAppender">
       <smtpHost>${smtpHost}</smtpHost>
       <smtpPort>${smtpPort}</smtpPort>
       <SSL>true</SSL>
       <username>${userName}</username>
       <password>${password}</password>
       <to>${to}</to>
       <from>${from}</from>

       <discriminator class="ch.qos.logback.classic.sift.MDCBasedDiscriminator">
           <key>req.remoteHost</key>
           <defaultValue>default</defaultValue>
       </discriminator>
       <subject>SMTP-WITH: %X{req.remoteHost} %logger{20} - %m</subject>

       <layout class="ch.qos.logback.classic.PatternLayout">
           <pattern>%date %-5level %logger{35} - %message%n</pattern>
       </layout>
   </appender>
```

다음은 MDCDiscriminator를 활용하는 샘플코드이다. 코드에서 직접 MDC에 동일한 키에 대해서 서로 다른값을 세팅하고 Error 로그 요청시 발송되게 처리하였다. 일반적인 웹환경에서는 사용자 요청 정보를 바탕으로 Servlet Filter를 사용하여 해당 Filter에서 필요한 키와 해당되는 값을 MDC에 값을 넣어서 로그를 분리하는 방법을 사용할 수 있다.

```
   String ip1 = "127.0.0.1";
   String ip2 = "255.255.255.255";

   MDC.put("req.remoteHost", ip1);
   logger.warn("the first message : " + ip1);

   //MDC에 저장된 값을 변경 다른 버퍼에 저장될수 있도록 함
   MDC.put("req.remoteHost", ip2);

   logger.warn("the first message : " + ip2);
   // 두번째 버퍼에 저장된 로그를 출력
   logger.error("the second message : " + ip2);
   // MDC에 저장된 값을 변경하여 최초에 저장된 버퍼에 로그가 저장되도록 함
   MDC.put("req.remoteHost", ip1);

   // 첫번째 버퍼에 저장된 로그를 출력
   logger.error("the second message : " + ip1);
```

해당 샘플코드를 실행하여 메일이 정상적으로 전송되면 두 건의 메일이 전송되고, 두 건의 메일내용이 각각 다른 MDC의 값에 분류된 내용이 전송이 된다. 다음은 해당 예제를 이용하여 전송된 메일 중 두번째 버퍼의 내용에 의해 전송된 메일 본문이다.

```
   2012-10-04 15:13:03,971 WARN  discriminatorSMTP - the first message : 255.255.255.255
   2012-10-04 15:13:03,971 ERROR discriminatorSMTP - the second message : 255.255.255.255
```

다음은 해당 예제를 이용하여 전송된 메일 중 첫번째 버퍼의 내용에 의해 전송된 메일 본문이다.

```
   2012-10-04 15:13:03,970 WARN  discriminatorSMTP - the first message : 127.0.0.1
   2012-10-04 15:13:03,971 ERROR discriminatorSMTP - the second message : 127.0.0.1
```

1.1.8.SyslogAppender

SyslogAppender의 경우 logging request를 Unix/Linux의 Syslog demon으로 전송하는 역할을 담당한다. SyslogAppender는 Syslog 프로토콜로 로그정보를 전송하는 역할을 하므로 SocketAppender 와 유사하게 별도의 Layout은 설정하지 않고 메시지는 포맷을 처리하기 위해서 suffixPattern을 사용한다.

SyslogAppender의 주요 속성은 다음과 같다.

속성명	기능	타입	기본값
syslogHost	syslog 서버 호스트	String	N/A
port	syslog 서버 포트	int	514
facility	메시지 소스를 식별하기 위한 식별자 KERN, USER, MAIL, DAEMON, AUTH, SYSLOG, LPR, NEWS, UUCP, CRON, AUTHPRIV, FTP, NTP, AUDIT, ALERT, CLOCK, LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7 중의 하나로 설정해야 한다.	String	
suffixPattern	syslog 서버로 전송될 메시지 포맷	String	[%thread] %logger %msg
throwableExcluded	Syslog 서버로 데이터 전송시 stack trace 데이터를 제외할지 여부	boolean	false

다음은 SyslogAppender를 설정하는 예제이다.

```
<appender name="syslog" class="ch.qos.logback.classic.net.SyslogAppender">
    <syslogHost>${syslogHost}</syslogHost>
    <port>${port}</port>
    <facility>KERN</facility>
    <suffixPattern>[%thread] %logger %msg</suffixPattern>
</appender>
```

SyslogAppender를 사용할 때 일반적으로 서버의 Syslog 데몬 설정은 디폴트로 remote의 log request를 처리하지 않도록 설정되어 있기 때문에 해당 설정을 변경해주어야 한다.

1.1.9.JMSQueue/JMSTopicAppender

JMSQueue/JMSTopicAppender는 JMS(JAVA Messaging Service)를 이용하여 로그메시지를 전송하는 역할을 담당한다. logback에서 제공하는 JMSAppender의 경우 내부적으로 Apache ActiveMQ를 이용하고 있기 때문에 해당 Appender를 사용하기 위해서는 ActiveMQ 라이브러리를 추가해야 한다. ActiveMQ 라이브러리 추가 시 주의할 점은 ActiveMQ 5.5 버전 부터 JDK 1.6 이상을 필요로 하기때문에 JDK 1.5를 사용하는 환경에서는 해당 버전보다 이전의 라이브러리를 사용해야 한다. 또한 JNDI를 사용하여 해당 Session과 Connection 객체 생성시에 사용하므로 클래스패스에 jndi.properties 파일이 필요하다.

다음은 JMSTopicAppender를 설정하는 예제이다. JMSQueueAppender는 전체적으로 JMSTopicAppender와 설정이 유사하고 일부 속성명 TopicConnectionFactoryBindingName을 QueueConnectionFactoryBindingName로 TopicBindingName을 QueueBindingName으로 변경하면 사용 가능하다.

```
<appender name="Topic" class="ch.qos.logback.classic.net.JMSTopicAppender">
    <InitialContextFactoryName>
    org.apache.activemq.jndi.ActiveMQInitialContextFactory
    </InitialContextFactoryName>
```

```
        <ProviderURL>${providerURL}</ProviderURL>
        <TopicConnectionFactoryBindingName>
           ConnectionFactory
        </TopicConnectionFactoryBindingName>
        <TopicBindingName>MyTopic</TopicBindingName>
    </appender>
```

1.1.10.SiftingAppender

Logback에서는 특정 런타임 기준에 따라 Appender를 분리하여 로깅을 수행할 수 있는 SiftingAppender를 제공한다. SiftingAppender는 기본적으로 MDC에 있는 값에 따라서 실제 사용하는 appender를 구분한다. appender가 서로 구분되어 생성되기 위해서는 서로 다른 name 속성을 가져야 하고 그렇게 하기 위해 일반적으로 MDC에 있는 값을 이용하여 이름을 설정한다. 일반적으로 SiftingAppender를 이용하는 경우는 FileAppender 혹은 RollingFileAppender를 이용하게 되는데 이때 파일이 생성되는 경로나 파일명도 appender 별로 구분될 수 있도록 설정한다.

다음은 SiftingAppender를 이용하여 MDC에 설정된 값에 따라 따라 서로 다른 파일에 로그를 남기게 한 예제이다.

```
<appender name="sift" class="ch.qos.logback.classic.sift.SiftingAppender">
    <discriminator>
        <key>mdcKey</key>
        <defaultValue>defaultValue</defaultValue>
    </discriminator>
    <sift>
        <appender name="${mdcKey}_appender"  class="ch.qos.logback.core.FileAppender">
            <file>${mdcKey}.log</file>
            ...
        </appender>
        ...
    </sift>
</appender>
```

1.2.Logger

Logger는 appender와 encoder를 이용하여 로깅을 수행하는 역할을 담당한다. 어플리케이션에서 Logger 객체를 검색할때 이름 규칙에 따라 가장 유사한 Logger 객체를 반환한다. 이때 일치되는 조건이 없는 경우 Root Logger를 반환한다.

Logger의 주요 속성은 다음과 같다.

속성명	기능	타입	기본값
name	해당 Logger의 고유한 식별자. Root Logger는 이름을 별도로 설정하지 않는다.	String	N/A
level	해당 Logger에 대한 로그 요청을 출력할지 여부를 결정하는 기준속성. 레벨은 TRACE<DEBUG<INFO<WARN<ERROR 이고 OFF로 설정하는 경우 모든 로그요청은 처리되지 않는다.	ch.qos.logback.classic.Level	N/A
additivity	해당 Logger의 상위 Logger에게 로그 이벤트를 전달할지 여부를 결정하는 속성	boolean	true

다음은 logger 와 Root를 설정하는 예제이다. 하나의 logger에 다중의 appender-ref를 설정할 수 있다.

```
 <logger name="myLogger" level="DEBUG" additivity="false">
    <appender-ref ref="console" />
```

```
        <appender-ref ref="file" />
    </logger>

    <root level="DEBUG">
        <appender-ref ref="console"/>
    </root>
```

1.3.Encoder

Encoder는 Layout과 pattern을 이용하여 로그이벤트 정보를 사용자가 설정한 형식에 맞춰서 변환하는 역할을 담당한다. encoder 설정 시에 클래스 속성을 설정하지 않는 경우 디폴트로 ch.qos.logback.classic.encoder.PatternLayoutEncoder가 사용된다.

다음은 encoder와 pattern을 사용하는 예제이다. 특별한 설정을 하지 않았기때문에 PatternLayoutEncoder가 사용된다. pattern 속성 설정시에는 %가 escape 문자로 사용된다. 샘플에 사용된 패턴은 %d로 현재 일시, %-5level 로그이벤트 요청시 레벨을 5글자 넓이로, %thread는 현재 쓰레드의 이름, %logger는 사용된 Logger의 풀네임과 로그 요청시 사용된 메시지를 출력하는 패턴을 지정하였다.

```
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>foo.log</file>
        <encoder>
            <pattern>%d %-5level [%thread] %logger: %msg%n</pattern>
        </encoder>
    </appender>
```

pattern에 지정하는 키워드에 대한 자세한 설명은 여기 [http://logback.qos.ch/manual/layouts.html#ClassicPatternLayout]를 참조한다.

1.4.StatusListener

Logback에서는 Logback 자체에서 발생하는 이벤트를 처리할 수 있도록 StatusListener 인터페이스를 제공한다. 해당 구현체로는 이벤트가 발생할때마다 상태 메시지를 콘솔로 출력해주는 OnConsoleStatusListener가 있다. 리스너를 사용하기 위해서는 설정파일에 해당 리스너를 등록해야 한다.

Logback에서는 로그를 수행하다가 발생하는 모든 Exception을 throw하지 않고 내부적으로 Error Status 객체를 생성하여 해당 정보를 담아서 기록한다. 따라서 로그를 남기는 중에 발생한 Exception을 처리하기 위해서는 Listener를 이용하면 일관된 방법으로 Exception을 처리할 수 있다.

다음은 Listener를 구현하여 Logback에서 발생하는 Exception을 처리하기 위한 예제이다.

```
    public class OnErrorStatusListener extends ContextAwareBase implements StatusListener,
LifeCycle {

    private boolean isStarted = false;

    public void addStatusEvent(Status status) {
        if (!isStarted()){
            return;
        }

        if (status.getLevel() == Status.ERROR){
            // 에러 처리
            ....
        }
    }
```

```
    public boolean isStarted() {
        return isStarted;
    }

    public void start() {
        isStarted = true;
    }

    public void stop() {
        isStarted = false;
    }
}
```

다음은 구현한 Listener를 설정파일에 등록하는 예제이다.

```
<statusListener class="org.anyframe.logback.appender.db.OnErrorStatusListener" />
```

1.5.Filter

Filter는 단순히 logger에 설정하는 level뿐 아니라 다양한 조건으로 로그 이벤트의 처리여부를 결정할 수 있다. 필터는 기본적으로 해당 로그 이벤트 처리여부를 결정하는 decide() 메소드를 가지고, 해당 메소드에서는 FilterReply를 enumeration을 리턴하는데 DENY, NEUTRAL, ACCEPT가 있다. 해당 필터가 DENY를 리턴하는 경우 해당 로그 이벤트는 처리가 중단되고, NEUTRAL인 경우 다음 필터가 존재하는 경우 다음 필터에서 처리여부를 결정하게 된다. ACCEPT가 리턴되는 경우 해당 로그 이벤트는 다음 필터에 관계 없이 처리되게 된다. 이러한 필터는 선언되어 있는 위치에 따라 전역적으로 사용되는 TurboFilter와 특정 Appender내에 존재하는 RegularFilter로 구분된다.

1.5.1.RegularFilter

RegularFilter는 특정 Appender의 내부에서 선언되어 Appender의 로그수행여부를 결정한다.

다음은 EvaluatorFilter와 JaninoEventEvaluator를 이용하여 SMTPAppender를 사용할때 MDC의 IP가 127.0.0.1인 경우는 로그를 남기지 않도록 한 설정파일 이다.

```
    <appender name="EMAIL_FILTER" class="ch.qos.logback.classic.net.SMTPAppender">

        <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
            <evaluator>
                <expression>return mdc.get("req.remoteHost") != null
                          && "127.0.0.1".equals(mdc.get("req.remoteHost")); </expression>
            </evaluator>
            <OnMismatch>NEUTRAL</OnMismatch>
            <OnMatch>DENY</OnMatch>
        </filter>

                <smtpHost>${smtpHost}</smtpHost>
        <smtpPort>${smtpPort}</smtpPort>
        <SSL>true</SSL>
        <username>${userName}</username>
        <password>${password}</password>
        <to>${to}</to>
        <from>${from}</from>
        <subject>SMTP-WITH: %logger{20} - %m</subject>
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern>%date %-5level %logger{35} - %message%n</pattern>
        </layout>
    </appender>
```

1.5.2.TurboFilter

TurboFilter는 기본적으로 RegularFilter와 기능적으로는 유사하지만 Context 전역적으로 선언되기때문에 모든 로그 이벤트 요청에 필터가 적용된다는 특징을 가지고 있다. Logback은 로그요청을 처리하는 가장 처음 단계로 TurboFilter를 적용한다. 반면에 Regular필터는 로그이벤트 객체를 생성하고 Logger 레벨을 확인하는 추가적인 과정을 거치기 때문에 전역적으로 필터링을 해야할 조건이 있는 경우 TurboFilter를 사용하면 성능상 이점을 얻을 수 있다. 단, TurboFilter를 사용할 때 주의점은 TurboFilter가 Accept를 리턴하는 경우 사용하려는 Logger의 레벨에 관계없이 해당 로그 이벤트가 처리된다는 특징이 있다는 점이다.

다음은 TurboFilter의 예제이다. MarkerFilter를 활용하여 billing Marker가 있는 경우 해당 로그 요청을 필터링 하도록 설정하였다.

```
<turboFilter class="ch.qos.logback.classic.turbo.MarkerFilter">
    <Marker>billing</Marker>
    <OnMatch>DENY</OnMatch>
</turboFilter>
```

1.6.MDC(Mapped Diagnostic Context)

Request 또는 사용자별로 특정한 정보를 로그로 남기기 위해서 사용할 수 있는것이 MDC이다. MDC는 Mapped Diagnostic Context의 약자로 쓰레드 별로 존재하는 Map과 유사한 객체이다. 하나의 request를 처리하기 위해서 thread를 사용하므로 이 MDC를 이용함으로써 사용자정보를 남길 수 있고 Logback은 MDC를 이용한 다양한 기능을 제공한다. 일반적인 web 환경의 경우 ServletFilter를 이용하여 해당 MDC의 정보를 관리하면 좀더 손쉽게 해당정보를 관리할 수 있다.

다음은 Logback Plugin 설치시 추가되는 MDCServletFilter 클래스이다. 요청처리 전 request와 session 정보를 이용하여 필요한 정보를 MDC에 넣고 요청이 처리가 완료되면 MDC의 값을 clear 한다.

```
public class MDCServletFilter implements Filter{

     public void doFilter(ServletRequest request, ServletResponse response,
         FilterChain chain) throws IOException, ServletException {
         insertIntoMDC(request);
         try{
             chain.doFilter(request, response);
         }finally{
             clearMDC();
         }
     }

     private void insertIntoMDC(ServletRequest request){
         MDC.put(REMOTE_HOST_MDC_KEY, request.getRemoteHost());

         if (request instanceof HttpServletRequest){
             HttpServletRequest httpServletRequest = (HttpServletRequest) request;

             StringBuffer requestURL = httpServletRequest.getRequestURL();
             if (requestURL != null) {
                 MDC.put(REQUEST_URL_MDC_KEY, requestURL.toString());
     }
             HttpSession session = httpServletRequest.getSession();

             if (session != null && session.getAttribute(USER_ID_MDC_KEY) != null){
                 MDC.put(USER_ID_MDC_KEY,
(String)session.getAttribute(USER_ID_MDC_KEY));
             }
```

```
            }
        }
    ...
```

주의할 점은 필터를 이용하여 MDC에 기록한 값은 처리가 request 처리가 끝난 후 반드시 clear를 해줘야 한다는 점이다.

MDCServletFilter를 구현한 후 web.xml에 등록해주어야 한다.

```
<filter>
    <filter-name>mdcServletFilter</filter-name>
    <filter-class>....logback.filter.MDCServletFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>mdcServletFilter</filter-name>
    <url-pattern>*.do</url-pattern>
</filter-mapping>
```

2.Features

Logback에서 새롭게 제공하는 기능은 다음과 같다. logback.xml과 같은 설정파일만을 수정함으로써 간단하게 기능을 적용할 수 있다.

- 설정파일의 Dynamic Reloading 지원
- 설정파일의 조건부 처리 기능
- 로그파일에 대한 자동압축, 자동 삭제 기능 제공
- 런타임에 설정한 값에 따라 로그를 분리하여 처리할 수 있는 SiftingAppender 제공
- groovy 언어로 설정파일 작성 기능(logback.groovy)
- FileAppender 사용 시 다수의 JVM이 동시에 하나의 파일에 로그를 남길 수 있는 prudent mode를 지원
- 다양한 조건에 따른 로깅처리 여부를 결정할 수 있는 Filter 제공

위 기능들 중 중요 기능에 대해서 좀 더 알아보도록 한다.

2.1.Dynamic Reloading

Logback은 자체적으로 설정 파일을 reloading 할 수 있는 기능을 제공한다. 설정파일의 <configuration>의 scan 속성과 scanPeriod 속성을 설정함으로써 Dynamic Reoloading을 설정할 수 있다. scanPeriod 속성을 명시하지 않는 경우 디폴트 값은 1분이 적용된다. scanPeriod에 적용할 수 있는 단위는 milliseconds, seconds, minutes, hours가 있고 단위를 생략하는 경우 경우 디폴트 단위는 milliseconds가 적용이 된다.

```
<configuration scan="true" scanPeriod="30 seconds">
    .....
</configuration>
```

주의할 점은 Logback은 로그 수행요청이 있을 경우에 설정파일에 명시한 시간간격을 체크 하여 설정파일을 reload 하므로, 로그 요청이 수행되어야 하고 로그가 호출되는 약간의 시간을 추가로 필요로 한다는 점이다.

2.2.Conditional Processing of Configuration Files

Logback은 하나의 설정파일을 다양한 조건에 따라 처리할 수 있는 <if>, <then>, <else> 태그를 제공한다. 해당 조건 태그들은 중첩해서 사용가능하고 설정파일의 어느위치에도 올 수 있다. 해당 기능을 사용하기 위해서 <if> 태그의 condition 속성에 expression을 설정해야하는데 expression을 작성하는 방식은 Java언어를 이용하는 방법과, groovy언어를 이용하는 방법이 존재한다. Java언어를 사용하기 위해서는 Janino 라이브러리(2.6 버전이상의 경우 Janino, commons-compiler), groovy언어를 사용하기 위해서는 groovy 런타임이 추가 라이브러리로 필요하다.

```
<if condition='java or groovy conditional expression'>
    <then>
        ...
    </then>
    <else>
        ...
    </else>
```

```
    </if>
```

2.3.Archiving

Logback에서는 RollingFileAppender를 사용하는 경우 로깅이 완료된 파일에 대해서 자동으로 압축하는 기능을 지원한다. 압축기능을 설정하기 위해서는 <fileNamePattern>에 확장자를 zip, gz로 설정하면 압축기능을 설정할 수 있다. 또한 TimeBasedRollingPolicy를 사용하는 경우 maxhistory 설정을 통해서 보관할 로그파일의 개수를 설정할 수 있고 오래된 파일을 자동으로 삭제해주는 기능을 지원한다.

```
    <appender name="file" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

            <fileNamePattern> someFileName.log.zip (or gz) </fileNamePattern>

              <maxHistory>90</maxHistory>
        </rollingPolicy>
        ....
    </appender>
```

fileNamePattern을 지정할 때 유의해야 하는 점은 날짜 형식을 파일명 혹은 폴더명으로 사용할 때 해당 OS지원하지 않는 특수문자가 포함되어서는 안된다는 점이다.

2.4.Logback Sample Configuration

Logback Plugin에 의해 설치된 logback.xml에서 앞서 언급한 새로운 기능들이 적용된 부분에 대해서 알아보고자 한다.

다음은 60초 간격으로 해당 설정 파일의 변경여부를 체크하고 변경된 설정파일을 적용하도록 한 설정이다.

```
    <configuration scan="true" scanPeriod="60 seconds">
        ...
    </configuration>
```

다음은 loggingCondition이라는 property를 이용하여 설정된 값에 따라 appender 설정을 변경하는 설정이다. 해당 appender에서는 로그 메시지와 함께 MDC에 있는 사용자 아이디, IP, 요청 URL 정보를 출력한다.

```
    <property name="loggingCondition" value="ID"/>
        <if condition='property("loggingCondition").equals("ID")'>
            <then>
                <!-- MDC에 저장된 사용자 아이디 값에 따라 로그를 분리하여 출력할 수 있는
SiftingAppender 설정-->
                <appender name="file" class="ch.qos.logback.classic.sift.SiftingAppender">
                    <discriminator>
                        <key>userId</key>
                        <defaultValue>guest</defaultValue>
                    </discriminator>
                    <sift>
                        <appender name="id-${userId}"
class="ch.qos.logback.core.FileAppender">
                            <file>userId/${userId}.log</file>
                            <encoder>
                                <pattern>[%-5level] %d{yyyy-MM-dd HH:mm:ss} %logger %n%msg%n
userId:%X{userId},clientIP:%X{remoteHost},requestURL:%X{requestURL}%n</pattern>
                            </encoder>
```

```
                        </appender>
                    </sift>
                </appender>
            </then>
            <else>
                <if condition='property("loggingCondition").equals("IP")'>
                    <then>
                    <!-- MDC에 저장된 IP값에 따라 로그를 분리하여 출력할 수 있는
 SiftingApppender 설정-->
                    <appender name="file"
 class="ch.qos.logback.classic.sift.SiftingAppender">
                        <discriminator>
                            <key>remoteHost</key>
                            <defaultValue>unknownHost</defaultValue>
                        </discriminator>
                        <sift>
                            <appender name="ip-${remoteHost}"
 class="ch.qos.logback.core.FileAppender">
                                <file>ip/${remoteHost}.log</file>
                                <encoder>
                                    <pattern>[%-5level] %d{yyyy-MM-dd HH:mm:ss} %logger %n
%msg%n userId:%X{userId},clientIP:%X{remoteHost},requestURL:%X{requestURL}%n</pattern>
                                </encoder>
                            </appender>
                        </sift>
                    </appender>
                </then>

                <else>
                    <!-- 프로퍼티가 정의되어있지 않거나 해당 조건에 없는경우 매초 로그를
 rolling하는 RollingFileAppender 설정 -->
                    <appender name="file"
 class="ch.qos.logback.core.rolling.RollingFileAppender">
                        <file>currentLog.log</file>
                        <rollingPolicy
 class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
                            <fileNamePattern>logs/logFile.%d{yyyy-MM-dd HH.mm.ss}.log.zip</
fileNamePattern>
                            <maxHistory>30</maxHistory>
                        </rollingPolicy>
                        <encoder>
                            <pattern>[%-5level] %d{yyyy-MM-dd HH:mm:ss} %logger %n%msg%n
 userId:%X{userId},clientIP:%X{remoteHost},requestURL:%X{requestURL}%n</pattern>
                        </encoder>
                    </appender>
                </else>
            </if>
        </else>
    </if>
```

3.Logging

본 장에서는 Logback을 이용하여 어플리케이션에서 로깅을 수행하는 방법에 대해서 설명하고자 한다.

3.1.기본적인 사용 방법

Logback은 slf4j를 구현하고 있기 때문에, 기본적으로 어플리케이션에서 로깅을 할때는 Logback에 종속적이지 않은 slf4j api만을 이용하여 로깅을 수행하는 것이 가능하다.

다음은 slf4j api를 이용하여 로그 메시지를 남기는 기본적인 예제이다.

```
import org.slf4j.Logger;
 import org.slf4j.LoggerFactory;

 public class LoggingTest {

     static Logger logger = LoggerFactory.getLogger(LoggingTest.class);

     public static void main(String[] args) {

         logger.info("the info message");
         logger.error("the error message");
     }
```

3.2.파라미터를 활용한 로깅

로그메시지를 동적으로 생성할 때 가장 간단한 방법은 문자열의 + 으로 문자열을 생성해서 전달하는 방법이다. 하지만 이 방법의 경우 해당 로그의 출력 여부와 관계없이 로그 이벤트 호출시 마다 매번 문자열 연산이 발생하므로 성능적인 면에서 비효율적이다. Logback을 포함한 Slf4j를 구현체들은 파라미터를 이용한 로깅 방식을 지원하여 불필요한 문자열 연산이 발생하지 않도록 한다.

다음은 하나의 파라미터를 이용하여 로그를 수행하는 예제이다. {} 문자를 이용하여 파라미터가 치환될 부분임을 명시할 수 있다.

```
    logger.error("Movie Information: movieId={}", movie.getMovieId());
```

다음은 두 개의 파라미터를 전달하는 예제이다.

```
    logger.debug("Movie Information: movieId={}, title={}" , movie.getMovieId(),
movie.getTitle());
```

세 개 이상의 파라미터를 전달하는 경우는 Object 배열을 활용해야 한다. 다음은 Object 배열을 이용하여 파라미터를 전달하는 예제이다.

```
    Object[] params =  new Object[]{movie.getMovieId(), movie.getTitle(),
movie.getReleaseDate()};
    logger.info("Movie Information: movieId={}, title={}, releaseDate={}", params);
```

3.3.Marker를 활용한 로깅

Marker는 로그를 남길 때 특정 정보를 추가적으로 전달하기 위해서 사용할 수 있는 객체이다. 특히 Logback에서 Marker를 활용한 Filter, EventEvaluator를 제공하는데 이 기능을 활용하기 위해서 Marker 객체가 활용된다.

다음은 어플리케이션에서 로그메시지와 함께 Marker를 전달하는 예제이다.

```
Marker marker = MarkerFactory.getMarker("MY_MARKER");
logger.info(marker, "marker example");
```

3.4.Exception 객체를 활용한 로깅

Exception이 발생한 경우 로그를 남길때 Exception 객체를 전달함으로써 Exception 정보를 전달하는것이 가능하다. 특히 Logback의 DBAppender를 활용할때 로그메시지와 Exception의 정보를 같이 전달하면 Exception 정보를 이용하여 LOGGING_EXCEPTION 테이블에 stacktrace 정보를 insert 한다.

다음은 어플리케이션에서 로그메시지와 함께 Exception 객체를 전달하는 예제이다.

```
try{
    ...
}catch(Exception ex){
    logger.error("An Exception Occured", ex);
}
```

III.References

- 참고자료
  - Logback [http://logback.qos.ch/]
  - MDC [http://logback.qos.ch/manual/mdc.html]
  - Groovy Configuration [http://logback.qos.ch/manual/groovy.html]