CODESTREET

Selector Framework
A generic data selection framework based on the JMS selector specification

A White Paper
Author: Jawaid Hakim
Published: January 2006
Last Updated: January 2009

Abstract

Applications frequently require the capability to select data based on flexible selection criteria. Usually the selection specification is custom built for each application. There is clear value in a flexible selection framework that is easy to use and has low performance impact on the application. An excellent example of a flexible selection specification is the selector specification in Java Message Service (JMS).

CodeStreet has leveraged the JMS selector specification to develop a generic selection framework that can be used to select arbitrary application data on the client side. The framework is called CodeStreet API for Data Selection (CADS). This framework is used in production quality systems at several top-tier Wall Street banks.

TABLE OF CONTENTS

1. Introduction

Message oriented middleware (MOM) has gained considerable currency as the preferred way to build highly decoupled and scalable systems. Many messaging protocols and their associated development frameworks exist in the marketplace. Some popular protocols and frameworks are TIBCO/Rendezvous[1], JMS[2], and SOAP[3].

JMS defines the concept of *selectors*. A selector is a conditional expression used by JMS clients to filter messages. A selector matches a message if the selector evaluates to *true* when the message's *headers* and *properties* are substituted for their corresponding identifiers in the selector[4].

Client applications can – optionally - specify a selector when they create a message consumer for a specific destination[5]. If a selector is specified for a message consumer, then all messages are filtered through the specified selector before being delivered to the consumer.

For example, suppose an application receives trade messages on a specific JMS queue and each trade message has two *long* properties - *Quantity* and *Price*. The following selector could be used to select messages where Quantity is greater than 100 and Price is less than 20:

```
String selector = “Quantity > 100 AND Price < 20”;
…
```

Applying selectors on the server (daemon) is the most efficient way to filter messages. However, most messaging systems and communication protocols do not provide built-in support for selectors.

An example of such a transport is TIBCO/RV. A TIBCO/RV client specifies interest in messages through subjects[6]. The RV daemon delivers all messages on the specified subject(s) to the client. Unlike JMS, TIBCO/RV clients cannot install a selector on the daemon to filter messages.

In fact, even in the case of JMS, if message selection is desired based on message *content* – not header fields and properties - then selectors must be applied on the client side.

[1] See http://www.tibco.com/solutions/products/active_enterprise/rv/default.jsp for details.
[2] See http://java.sun.com/products/jms for details.
[3] See http://www.w3.org/TR/SOAP/ for details.
[4] A JMS message consists of three distinct sections – headers fields, properties, and application fields. Header fields are set by the JMS daemon – e.g. JMSMessageID. Properties are set by the application or the specific JMS implementation – e.g. JMS_TIBCO_MSG_EXT. Applications fields are set by the application.
[5] A JMS destination is either a topic (one-to-many) or a queue (one-to-one).
[6] A TIBCO/RV message send subject is analogous to a JMS destination.

This white paper describes CADS, a general purpose and highly efficient framework for implementing conditional expression evaluation functionality in applications. The framework described here is a complete implementation of the JMS selector specification 1.1.

2. Message Selector Syntax

A message selector is a String whose syntax is based on a subset of the SQL92[7] conditional expression syntax.

The order of evaluation of a selector is from left to right within precedence level. Parenthesis can be used to change the evaluation order.

Predefined selector literals and operator names are shown here in uppercase, however, they are case insensitive.

A selector can contain:

- Literals:
  - A string literal is enclosed in single quotes.
  - An exact numeric literal is a numeric value without a decimal point, such as 57, -91, and +62; numbers in the range of Java *long* as supported.
  - An approximate numeric literal is a numeric value is scientific notation, such as 7E3, -57.9E2, or a numeric value with a decimal such as 7., and -95.6; numbers is the range of Java *double* are supported.
  - The boolean literals *TRUE* and *FALSE*.
- Identifiers:
  - An identifier is an unlimited-length character sequence must begin with a Java identifier start character, all following characters must be Java identifier part characters.
  - Identifier cannot be the names *NULL*, *TRUE*, *FALSE*, *NOT*, *IN*, *BETWEEN*, *AND*, *OR*, *IS*, or *ESCAPE*.
  - Identifiers are either header field references or property references. If a property that does not exist is referenced, its value is NULL.
  - No type conversions are supported. For example, suppose a message contain a *String* property *Address*. If Address is used in an arithmetic sub-expression, then the sub-expression evaluates to FALSE.
  - Identifiers are case sensitive.
  - Message header fields are restricted to *JMSDeliveryMode*, *JMSPriority*, *JMSMessageID*, *JMSTimestamp*, *JMSCorrelationID*, and *JMSType*.

[7] See X/OPEN CAE Specification Data Management: Structured Query Language (SQL), Version 2, ISBN: 1-85912-151-9 March 1996.

JMSMessageID and JMSType values may be *null* and if so are treated as a NULL value.

- Whitespace is the same as defined for Java: space, horizontal tab, form feed and line terminator.
- Expressions:
  - A selector is a conditional expression; a selector that evaluates to *true* matches; a selector that evaluates to *false* or *unknown* does not match.
  - Arithmetic expressions are composed of themselves, arithmetic operators, identifiers with numeric values, and numeric literals.
  - Conditional expressions are composed of themselves, comparison operators, logical operations, identifiers with boolean values, and boolean literals.
- Standard bracketing () for ordering expression evaluation is supported.
- Logical operators in precedence order: *NOT*, *AND*, *OR*.
- Comparison operators: =, >, >=, <, <=, <> (not equal)
  - Only like type values can be compared. If comparison on non-like type values is attempted, the value of the operation is *FALSE*. If either of the type values evaluates to *NULL*, the value of the expression is *UNKNOWN*.
  - *String* and *Boolean* comparison is restricted to = and <>.
- Arithmetic operators in precedence order:
  - +, - (unary)
  - *, / (multiplication and division)
  - +, - (addition and subtraction)
- *arith-expr1* [NOT] BETWEEN *arith-expr2* AND *arith-expr3* (comparison operator)
  - "Age BETWEEN 15 AND 19" is equivalent to "Age >= 15 AND Age <= 19".
  - "Age NOT BETWEEN 15 AND 19" is equivalent to "Age < 15 OR Age > 19".
- *identifier* [NOT] IN (*str-literal1*, *str-literal2*, …) (comparison operator where identifier has a *String* or NULL value)
  - "Country in ('US', 'UK', 'France')" is equivalent to "Country = 'US' OR Country = 'UK' OR Country = 'France'",
- *identifier* [NOT] LIKE *pattern-value* [ESCAPE *escape-character*] (comparison operator where identifier has a *String* value; pattern-value is a string literal where '_' stands for any single character; '%' stands for any sequence of characters, including the empty sequence, and all characters stand for themselves. The optional *ESCAPE* character is a single-character string literal whose character is used to escape the special meaning of '_' and '%' in *pattern-value*.)
  - "Phone LIKE 718%" is true for '718' or '718123' and false for '719'.
  - "Name LIKE j_n" is true for 'jon' or 'jan' and false for 'john'.
- *identifier* IS NULL (comparison operator that checks for null header field value or missing property value).

- *identifier* IS NOT NULL (comparison operator that checks for the existence of a non-null header field value or property value).

1.1. Null Values

As noted above, header fields and property values may be NULL. The evaluation of selector expressions containing NULL values is defined by SQL92 NULL semantics. A brief description is provided here.

SQL treats a NULL value as *unknown*. Comparison or arithmetic with an unknown value always yields an unknown value. The IS NULL and IS NOT NULL operators convert an unknown header field or property into the respective TRUE and FALSE values.

3. CADS API

With the selector syntax background under our belts, it is time to get a look at the CADS API. The API can be divided into two groups: the API for creating selectors and the API for evaluating selectors.

1.2. Creating Selectors

A selector within CADS is created as follows:

```
try
{
    // Create selector
    String expr = “Quantity > 100 AND Price < 20”;
    ISelector isel = Selector.getInstance(expr);
    …
}
catch (InvalidSelectorException ex)
{
    …
}
```

Behind the scene, the specified selector expression is parsed using a high-speed JAVACC[8] parser, and an in-memory parse tree is built for efficient evaluation.

1.3. Evaluating Selectors

Once a *Selector* instance has been successfully created, there are two options available for evaluation of the selector. This section describes the two options in detail.

1.1.1. Value Map

[8] See http://www.experimentalstuff.com/Technologies/JavaCC/ for details.

In the first option, the application *asks* the selector for the list of identifiers that were encountered during the parse. The application then creates a *Map* with values for each identifier. The application then evaluates the selector using the Map:

```
try
{
    // Create selector
    String expr = “Quantity > 100 AND Price < 20”;
    ISelector isel = Selector.getInstance(expr);

    // Get identifiers encountered during the parse
    Map identifiers = isel.getIdentifiers();

    // Specify values for identifiers
    Map values = new HashMap();
    for (Iterator iter = identifiers.values().iterator();
iter.hasNext(); )
    {
        // Set value for identifier
        Identifier id = (Identifier)iter.next();
        String idName = id.getIdentifier();
        values.put(idname, getValueForId(idname));
    }

    // Evaluate
    Result result = isel.eval(values);

    …
}
catch (InvalidSelectorException ex)
{
    …
}
```

Note, the method *getValueForId()* is application code.

1.1.2. Value Provider

The second option is based on the notion of *value provider* classes. The idea is to allow applications to register a value provider with the selector. During evaluation of the selector, the selector invokes *callbacks* on the registered provider to get the value of the desired identifier. This option is more efficient because the value provider is invoked only for values that are actually required to evaluate the expression. For example, if the selector expression consists of two sub-expressions connected by an OR and the left-hand side sub-expression evaluated to TRUE, then the second sub-expression is not evaluated.

The following code fragment shows how a value provider is used. The actual implementation of *IValueProvider* that is used will depend on the application.

The framework contains built-in value provider classes for Java Beans, JMS, and TIBCO/RV. For extreme performance, caching value provider implementations are provided for Java Beans and JMS. Caching value identifier values can provide a significant performance boost.

```
import com.codestreet.selector.Selector;
import com.codestreet.selector.parser.IValueProvider;
import com.codestreet.selector.parser.InvalidSelectorException;
import com.codestreet.selector.parser.Result;
import
com.codestreet.selector.provider.bean.CachingValueProvider;

try
{
    // Create selector
    String expr = “Quantity > 100 AND Price < 20”;
    ISelector isel = Selector.getInstance(expr);

    // Get appropriate value provider
    IValueProvider vp = …;

    // Get correlation data (or NULL) required by value provider
    Object corr = …;

    // Evaluate
    Result result = isel.eval(vp, corr);

    …
}
catch (InvalidSelectorException ex)
{
    …
}
```

Applications are free to implement their own value providers. Implementing a value provider is as simple as coding a class that implements the *IValueProvider* interface.

1.4. Selector Caching

CADS does automatic selector caching. Without this caching, it would be easy for clients to use CADS inefficient. This is because selector *evaluation* is much faster than selector *creation*.

Without caching, the following code shows how **not** to use CADS:

```
try
{
    for (int i = 0; i < numMsgs; ++i)
    {
        // Create selector
        String selector = “Quantity > 100 AND Price < 20”;
        ISelector isel = Selector.getInstance(selector);

        IValueProvider vp = …;
        Object corr = …;

        // Evaluate
```

```
                Result result = isel.eval(vp, corr);

                …
            }
        }
        catch (InvalidSelectorException ex)
        {
            …
        }
```

What is wrong with the code shown above? The selector is created inside the loop, even though the selector is unchanged through the loop. The code shown above works but is highly inefficient.

A better usage is the following:

```
        try
        {
            // Create selector
            String expr = “Quantity > 100 AND Price < 20”;
            ISelector isel = Selector.getInstance(expr);

            for (int i = 0; i < numMsgs; ++i)
            {
                IValueProvider vp = …;
                Object corr = …;

                // Evaluate
                Result result = isel.eval(vp, corr);

                …
            }
        }
        catch (InvalidSelectorException ex)
        {
            …
        }
```

CADS, in release 2.5 and higher, caches selectors in a concurrent hash map. As a result, both code snippets shown above will provide almost identical performance. If you do not want selector caching then you can clear the cache explicitly:

```
        try
        {
            // Create selector
            String expr = “Quantity > 100 AND Price < 20”;
            ISelector isel = Selector.getInstance(expr);
            Selector.clear();

            for (int i = 0; i < numMsgs; ++i)
            {
                IValueProvider vp = …;
                Object corr = …;

                // Evaluate
                Result result = isel.eval(vp, corr);
                …
            }
        }
        catch (InvalidSelectorException ex)
        {
            …
        }
```

4. JMS Extensions

The JMS specification restricts selector identifiers to header fields and properties values. However, applications frequently require the ability to filter messages based on their content. One option for applications is to publish relevant message fields as properties in order to allow selection. This requires changes to the application code.

To allow applications to filter messages based on content, the framework provides the following extension to the JMS specification: identifier names can contain '.' (dot). This extension can be used by value provider classes to extract values of message content fields.

CADS value provider classes for JMS and TIBCO/RV use the dot notation to provide content-based selection. These two classes treat any identifier prefixed with a '.' as a message content field.

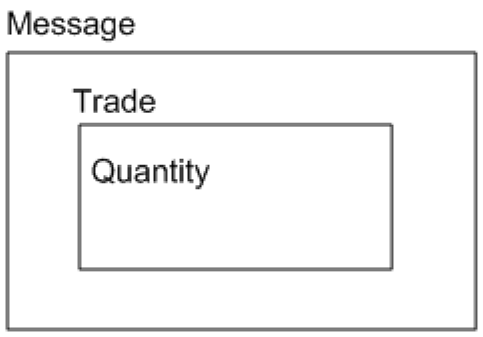
For example, the following code selects JMS messages based on the value of a content field called *Quantity*:

```
    try
    {
        // Create selector
        String expr = ".Quantity > 100";
        ISelector isel = Selector.getInstance(expr);

        // Get the JMS message for filtering
        Message msg = …;
        // Get appropriate value provider
        IValueProvider vp = new
    com.codestreet.selector.jms.ValueProvider(msg);

        // Get correlation data (or null) required by value provider
        Object corr = null;

        // Evaluate
        Result result = isel.eval(vp, corr);
        if (result == Result.UNDEFINED)
        {
            …
        }
        else if (result == Result.TRUE)
        {
            …
        }
        else
        {
            // FALSE
            …
        }

        …
    }
    catch (InvalidSelectorException ex)
    {
        …
    }
```

1.5. Nested Messages

Some JMS vendors provide allow nested messages within messages. TIBCO/JMS is an example of such an implementation.

Suppose a JMS message contains a nested message named *Trade*. The *Trade* message in turn contains a field named *Quantity*. The diagram below shows the conceptual structure of this message:



The following code uses the value provider class for JMS to select JMS messages based on the value of *Quantity*:

```
        try
        {
            // Create selector
            String expr = ".Trade.Quantity > 100";
            ISelector isel = Selector.getInstance(expr);

            // Get the JMS message for filtering
            Message msg = …;
            // Get appropriate value provider
            IValueProvider vp = new
        com.codestreet.selector.jms.ValueProvider(msg);

            // Get correlation data (or NULL) required by value provider
            Object corr = null;

            // Evaluate
            Result result = isel.eval(vp, corr);
            if (result == Result.UNDEFINED)
            {
                …
            }
            else if (result == Result.TRUE)
            {
                …
            }
            else
            {
                // FALSE
                …
            }

            …
        }
        catch (InvalidSelectorException ex)
        {
            …
        }
```

5. Thread Safety

The framework makes heavy use of the immutable[9] pattern. All classes in the framework are immutable. As a result, no internal or external synchronization is required. Instances of *CSSelector* may be shared freely between threads.

6. Performance

CADS is a high performance framework with low memory usage. Once a selector has been parsed and the in-memory representation has been built, evaluating it is extremely efficient.

For example, consider the following selector:

```
    String selector = "JMSPriority >= 0 AND Quantity > 100 AND
MessageName in ('Login', ‘Logout’, 'Query', 'CreateOrder',
'CreateWatchlist', 'Trade', 'OrderAction', 'Orderbook’)";
```

This selector can be evaluated by CADS – using the JMS value provider - at a rate of about 200,000 evaluations per second on a low-end PC running Windows 2000 with 512 MB of RAM.

As a comparison, the leading open source selector implementation - in JBOSS[10] - is roughly *four* times slower.

[9] The easiest way to make a class thread-safe is to make it immutable. Since an immutable objects state is set during construction and cannot be changed afterwards, the object can be freely shared between threads without synchronization. See Effective Java by Joshua Bloch for more details, ISBN: 0-201-31005-8.
[10] See http://www.jboss.org

7. Support for .NET

CADS is also available as a .NET framework. The .NET framework has been written from the ground up in C#. Any .NET/CLS-compliant language can use the framework.

8. Summary

CADS is a complete implementation, along with useful extensions, of the JMS selector specification. It can be used in any context – JMS, TIBCO/RV, etc. - where conditional expression evaluation is required.

For further information contact:

Jawaid Hakim
CTO
jawaid.hakim@codestreet.com