Messaging Mapper

Overview

The *messaging mapper* pattern describes how to map domain objects cleanly to and from a canonical message format.

The purpose of the messaging mapper pattern is to create a clean mapping from domain objects to a canonical message format, where the message format is chosen to be as platform neutral as possible. In other words, the chosen message format should be suitable for transmission through a message bus on page 50, where the message bus is the backbone for integrating a variety of different systems, some of which might not be object-oriented.

Many different approaches are possible, but not all of them are clean enough to fulfill the requirements of a messaging mapper. For example, an obvious way to transmit an object would be to use *object serialization*, which enables you to write an object to a data stream using an unambiguous encoding (supported natively in Java). This would *not* be a suitable approach to use for the messaging mapper pattern, however, because the serialization format is understood only by Java applications. Java object serialization would create an impedance mismatch between the original application and the other applications in the messaging system.

The requirements on a messaging mapper can be summarized as follows:

- The canonical message format used to transmit domain objects should be suitable for consumption by non-object oriented applications.
- The mapper code should be implemented separately from the domain object code and separately from the messaging infrastructure. FUSE Mediation Router helps you to fulfill this requirement by providing hooks that can be used to insert mapper code into a route.
- The mapper might need to find an effective way of dealing with certain object-oriented concepts such as inheritance, object references, and object trees. The complexity of these issues will vary from application to application, but the aim of the mapper implementation must always be to create messages that can be processed effectively by non-object-oriented applications.

Finding objects to map

You could use one of the following mechanisms to find the objects to map:

- *Find a registered bean*—for singleton objects and small numbers of objects, you could use the `CamelContext` registry to store references to beans. For example, if a bean instance is instantiated using Spring XML, it is automatically entered into the registry, where the bean is identified by the value of its `id` attribute.

- *Select objects using the JoSQL language*—if all of the objects you want to access are already instantiated at runtime, you could use the JoSQL language to locate a specific object (or objects). For example, if you have a class, `org.apache.camel.builder.sql.Person`, with a `name` bean property and the incoming message has a `UserName` header, you could select the object whose `name` property equals the value of the `UserName` header using the following code:

```
// Java
import static org.apache.camel.builder.sql.SqlBuilder.sql;
import org.apache.camel.Expression;
...
Expression expression = sql("SELECT * FROM
org.apache.camel.builder.sql.Person where name = :UserName");
Object value = expression.evaluate(exchange);
```

  Where the syntax, `:`*`HeaderName`*, is used to substitute the value of a header in a JoSQL expression.

- *Dynamic*—for a more scalable solution, it might be necessary to read object data from a database. In some cases, the existing object-oriented application might already provide a finder object that can load objects from the database. In other cases, you might have to write some custom code to extract objects from a database: the JDBC component and the SQL component might be useful in these cases.