Custom Message Generation - Packaging

Overview

As of QFJ 1.2.0, there is additional support for custom message generation. This article discusses how custom messages can be generated into user-specified packages.

> This is an advanced feature and the support for it is currently limited. However, with user feedback we can provide increasingly better tools to support this feature if enough users need it.

The steps for generated messages into custom packages are:

1. Create a customized data dictionary XML file.
2. Create a program to generate messages using `quickfix.codegen.MessageGenerator`.
3. Build a JAR file containing the compiled generated classes

The code for this example is attached to this topic (See below).

File	Modified
qfj-custom.zip Example project for building custom message JAR	Jul 04, 2007 by Steve Bate

Create a Custom Data Dictionary

For the example in this article, the customized data dictionary file is just a copy of FIX44.xml. This file could then be modified for your specific purposes. You must be sure all referenced components and fields are present to support the existing or new messages you have defined.

The basic structure of the data dictionary XML file is shown below.

- Message header
- Message trailer
- Messages
- Components
- Fields

You will define new messages (or include standard messages) in the messages section. All components referenced by the messages must be present in the components section. All fields referenced by either messages or components must be in the fields section.

For the example described in this article, we'll create a project directory to generate our custom message library. We'll name our custom data dictionary, `spec/CustomFIX.xml`.

Create a Message Generation Driver Program

Now we'll create a program to generate our custom classes. It will be called `custom.MyCodeGenerator` and placed in a `src` directory.

```
package custom;

import quickfix.codegen.MessageCodeGenerator;
import quickfix.codegen.MessageCodeGenerator.Task;

public class MyCodeGenerator {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.err.println("Must supply base package name");
        }
        String basePackage = args[0];
        Task task = new Task();
        task.setName("Custom 4.4");
        task.setSpecification("spec/CustomFIX.xml");
        task.setTransformDirectory("transforms");
        task.setOutputBaseDirectory("src-generated");
        task.setMessagePackage(basePackage + ".messages");
        task.setFieldPackage(basePackage + ".fields");
        MessageCodeGenerator codeGenerator = new MessageCodeGenerator();
        codeGenerator.generate(task);
    }
}
```

The class requires an argument specifying the base package name. This could also be hardwired, but it's convenient for the Ant script you'll see later. The QFJ code generation class generates messages based on a `Task` definition. The task definition we are using specifies:

- Name - informational only, used for logging output.
- Specification - the path to our custom FIX data dictionary.
- TransformDirectory - Where XSLT transforms are stored
- OutputBaseDirectory - Base directory for writing generated classes
- MessagePackage - the package for messages, components, and supporting classes like factories.
- FieldPackage - the package where fields are generated

In our example, we have XSLT transforms in the "transforms" subdirectory and will generate classes into "src-generated". The Ant script will specify "custom" as the base package name so our messages will be in `custom.messages.` **and fields in `custom.fields.`**. The directory structure looks like:

- src/
  - custom/
    - MyCustomCodeGenerator
- spec/
  - CustomFIX.xml
- transforms/
  - Fields.xsl (and others)
- lib/
  - quickfixj-core.jar (and other dependencies)

If you have multiple data dictionary files you can create a task for each one in your driver program. See the main function of the QFJ `MessageCodeGenerator` for an example of doing this.

Build a Custom JAR

The Ant script for building the custom message JAR file will compile our code generator class, generate the message code, compile the message code, and create the JAR file. The JAR file will have the message classes and the CustomFIX.xml data dictionary. In the example below, I've also include a target to remove the generated files and to compile a modified `Executor` program (from the QFJ examples) that uses the custom message library.

```
<project name="qfj-custom">
 <property name="base.package" value="custom" />
 <property name="classes.dir" value="classes" />
 <property name="generated.src.dir" value="src-generated" />
 <property name="generated.classes.dir" value="classes-generated" />
 <property name="specification.dir" value="spec" />
 <property name="jar.name" value="qfj-custom-msg.jar" />

 <path id="classpath">
  <pathelement location="${classes.dir}" />
  <pathelement location="${generated.classes.dir}" />
  <fileset dir="lib" includes="*.jar" />
 </path>

 <target name="compile.generator">
  <mkdir dir="${classes.dir}"/>
  <javac srcdir="src" destdir="${classes.dir}"
   includes="custom/**/*"
   classpathref="classpath" />
 </target>

 <target name="generate" depends="compile.generator">
  <java classname="custom.MyCodeGenerator" fork="true"
classpathref="classpath">
   <arg value="${base.package}" />
  </java>
 </target>

 <target name="compile" depends="generate">
  <mkdir dir="${generated.classes.dir}" />
  <javac srcdir="${generated.src.dir}"
   destdir="${generated.classes.dir}"
   classpathref="classpath"
   fork="true" memorymaximumsize="512m"/>
 </target>

 <target name="jar" depends="compile" description="Create custom message
JAR">
  <jar destfile="${jar.name}">
   <fileset dir="${generated.classes.dir}" />
   <fileset dir="${specification.dir}" />
  </jar>
 </target>

 <target name="clean" description="Remove generated project files">
  <delete dir="classes" />
  <delete dir="src-generated/${base.package}" />
  <delete dir="classes-generated" />
  <delete file="${jar.name}" />
 </target>

 <target name="compile.examples" description="Compile example code">
```

```
 <javac srcdir="src" destdir="${classes.dir}"
  includes="quickfix/**/*"
  classpathref="classpath" />
</target>
```

```
</project>
```

To demonstrate the usage of the custom message library, the example project includes a modified version of the `Executor` program. The modifications are minimal and simple.

- Use the message factory from custom library
- Specify the custom data dictionary in the settings file

```
...
  MessageFactory messageFactory = new custom.messages.MessageFactory();

  acceptor = new SocketAcceptor
    (application, messageStoreFactory, settings, logFactory,
messageFactory);
```

```
[default]
FileStorePath=examples/target/data/executor
DataDictionary=CustomFIX.xml
ConnectionType=acceptor
...
```

Potential Issues

If you have a connector (acceptor, initiator) that will have some sessions using standard data dictionaries and some using custom dictionaries, then you will need to hand write a custom message factory. One option is to extend the `DefaultMessageFactory` included in QFJ. However, this may be difficult to implement successfully if the standard and custom data dictionaries are for the same version of FIX since the default factory uses the version for delegating to a version-specific message factory. If you are in this situation, you may need to write a message factory that delegates to other factories based on different criteria, like the session identifier (for example).