Type Converter

Its very common when routing messages from one endpoint to another to need to convert the body payloads from one type to another such as to convert to and from the following common types

- File
- String
- byte[] and ByteBuffer
- InputStream and OutputStream
- Reader and Writer
- Document and Source
- ...

The Message interface defines a helper method to allow conversions to be done via the getBody(Class) method.

So in an endpoint you can convert a body to another type via

```
Message message = exchange.getIn();
Document document = message.getBody(Document.class);
```

How Type Conversion works

The type conversion strategy is defined by the TypeConverter interface that can be customized on a CamelContext.

The default implementation, DefaultTypeConverter, uses pluggable strategies to load type converters via TypeConverterLoader. The default strategy, AnnotationTypeConverterLoader, uses a discovery mechanism to find converters.

New in Camel 1.5

The default implementation, DefaultTypeConverter, now throws a NoTypeConversionAvailableException if a suitable conversion cannot be found (CAMEL-84). The semantical ambiguity of `null` (both valid result and indication of no conversion found) is now resolved, but this may impact existing code in that it should now catch the exception instead of checking for `null`.

TypeConverterRegistry

New in Camel 2.0

Exposed the TypeConverterRegistry from CamelContext so end users more easily will be able to add type converters at runtime. This is also usable in situations where the default discovering of type converters fails on platforms with classloading issues.

To access the registry, you get it from the CamelContext

```
CamelContext context = ...
context.getTypeConverterRegistry()
```

TypeConverterRegistry utilization statistics

Camel can gather utilization statistics of the runtime usage of type converters. These stats are available in JMX, and as well as from the `getStatistics()` method from `TypeConverterRegistry`.

From **Camel 2.11.0/2.10.5** onwards these statistics are turned off by default as there is some performance overhead under very high concurrent load. To enable the statistics in Java, do the following:

```
CamelContext context = ...
context.setTypeConverterStatisticsEnabled(true);
```

Or in the XML DSL with:

```
<camelContext xmlns="http://camel.apache.org/schema/spring"
typeConverterStatisticsEnabled="true">
...
</camelContext>
```

Add type converter at runtime

The following sample demonstrates how to add a type converter at runtime:

```
// add our own type converter manually that converts from String -> My
MyOrderTypeConverter
context.getTypeConverterRegistry().addTypeConverter(MyOrder.class, Str
MyOrderTypeConverter());
```

And our type converter is implemented as:

```
private static class MyOrderTypeConverter extends TypeConverterSupport

    @SuppressWarnings("unchecked")
    public <T> T convertTo(Class<T> type, Exchange exchange, Object va
        // converter from value to the MyOrder bean
```

```
        MyOrder order = new MyOrder();
        order.setId(Integer.parseInt(value.toString()));
        return (T) order;
    }
}
```

And then we can convert from String to MyOrder as we are used to with the type converter:

```
MyOrder order = context.getTypeConverter().convertTo(MyOrder.class, "
```

Add type converter classes at runtime

Available as of Camel 2.16

From Camel 2.16 onwards you type converter classes can implement `org.apache.camel.TypeConverters` which is an marker interface. Then for each type converter you want use the `@Converter` annotation.

```
private class MyOrderTypeConverters implements TypeConverters {

    @Converter
    public MyOrder toMyOrder(String orderId) {
        MyOrder order = new MyOrder();
        order.setId(Integer.parseInt(orderId));
        return order;
    }
}
```

Then you can add these converters to the registry using

```
MyOrderTypeConverters myClass = ...
context.getTypeConverterRegistry().addTypeConverters(myClass);
```

If you are using Spring or Blueprint, then you can just declare a <bean> then CamelContext will automatic discover and add the converters.

```
<bean id="myOrderTypeConverters" class="..."/>

<camelContext ...>
   ...
</camelContext>
```

You can declare multiple <bean>s if you have more classes.

Using this technique do not require to scan the classpath and using the file `META-INF/services/org/apache/camel/TypeConverter` as discussed in the *Discovering Type Converters* section. However the latter is highly recommended when developing Camel components or data formats as then the type converters is automatic included out of the box. The functionality from this section requires the end users to explicit add the converters to their Camel applications.

Discovering Type Converters

The AnnotationTypeConverterLoader will search the classpath for a file called *META-INF/services/org/apache/camel/TypeConverter*. The contents are expected to be comma separated package names. These packages are then recursively searched for any objects with the @Converter annotation. Then any method marked with @Converter is assumed to be a conversion method; where the parameter is the from value and the return is the to value.

e.g. the following shows how to register a converter from File -> InputStream

```
@Converter
public class IOConverter {

    @Converter
    public static InputStream toInputStream(File file) throws FileNotFo
        return new BufferedInputStream(new FileInputStream(file));
    }
}
```

Static methods are invoked; non-static methods require an instance of the converter object to be created (which is then cached). If a converter requires configuration you can plug in an Injector interface to the DefaultTypeConverter which can construct and inject converter objects via Spring or Guice.

We have most of the common converters for common Java types in the org.apache.camel.converter package and its children.

Returning null values

By default when using a method in a POJO annotation with @Converter returning null is not a valid response. If null is returned,

then Camel will regard that type converter as a *miss*, and prevent from using it in the future. If null should be allowed as a valid response, then from **Camel 2.11.2/2.12** onwards you can specify this in the annotation as shown:

```
@Converter(allowNull = true)
public static InputStream toInputStream(File file) throws IOException {
    if (file.exist()) {
        return new BufferedInputStream(new FileInputStream(file));
    } else {
        return null;
    }
}
```

Discovering Fallback Type Converters

Available in Camel 2.0

The AnnotationTypeConverterLoader has been enhanced to also look for methods defined with a @FallbackConverter annotation, and register it as a fallback type converter.

Fallback type converters are used as a last resort for converting a given value to another type. Its used when the regular type converters give up.
The fallback converters is also meant for a broader scope, so its method signature is a bit different:

```
@FallbackConverter
public static <T> T convertTo(Class<T> type, Exchange exchange, Object
TypeConverterRegistry registry)
```

Or you can use the non generic signature.

```
@FallbackConverter
public static Object convertTo(Class type, Exchange exchange, Object va
TypeConverterRegistry registry)
```

And the method name can be anything (`convertTo` is not required as a name), so it can be named `convertMySpecialTypes` if you like.
The `Exchange` parameter is optional, just as its with the regular `@Converter` methods.

The purpose with this broad scope method signature is allowing you to control if you can convert the given type or not.
The `type` parameter holds the type we want the `value` converted to.

Its used internally in Camel for wrapper objects so we can delegate the type convertions to the body that is wrapped.

For instance in the method below we will handle all type conversions that is based on the wrapper class `GenericFile` and we let Camel do the type conversions on its body instead.

```
@FallbackConverter
public static <T> T convertTo(Class<T> type, Exchange exchange, Object
TypeConverterRegistry registry) {
    // use a fallback type converter so we can convert the embedded b
GenericFile
    if (GenericFile.class.isAssignableFrom(value.getClass())) {
        GenericFile file = (GenericFile) value;
        Class from = file.getBody().getClass();
        TypeConverter tc = registry.lookup(type, from);
        if (tc != null) {
            Object body = file.getBody();
            return tc.convertTo(type, exchange, body);
        }
    }

    return null;
}
```

Writing your own Type Converters

Use FQN

In **Camel 2.8** the TypeConverter file now supports specifying the FQN class name. This is recommended to be used. See below for more details

You are welcome to write your own converters. Remember to use the @Converter annotations on the classes and methods you wish to use. Then add the packages to a file called *META-INF/services/org/apache/camel/TypeConverter* in your jar. Remember to make sure that :-

- static methods are encouraged to reduce caching, but instance methods are fine, particularly if you want to allow optional dependency injection to customize the converter
- converter methods should be thread safe and reentrant

Examples of TypeConverter file

The file in the JAR: `META-INF/services/org/apache/camel/TypeConverter` contains the following line(s)

```
com.foo
com.bar
```

Each line in the file is a package name. This tells Camel to go scan those packages for any classes that has been annotated with the @Converter.

Improved TypeConverter by using FQN class names

Available as of Camel 2.8

In Camel 2.8 we improved the type converter loader to support specifying the FQN class name of the converter classes. This has the advantage of avoiding having to scan packages for @Converter classes. Instead it loads the @Converter class directly. This is a **highly** recommend approach to use going forward.

Examples of TypeConverter file

The file in the JAR: `META-INF/services/org/apache/camel/TypeConverter` contains the following line(s) for FQN class names

```
com.foo.MyConverter
com.bar.MyOtherConverter
com.bar.YetOtherConverter
```

As you can see each line in the file now contains a FQN class name. This is the recommended approach.

Encoding support for byte[] and String Conversion

Available in Camel 1.5

Since Java provides converting the byte[] to String and String to byte[] with the charset name parameter. You can define the charset name by setting the exchange property name `Exchange.CHARSET_NAME` with the charset name, such as `"UTF-8"` or `"iso-8859-1"`.

Exchange parameter

Available in Camel 1.5

The type converter accepts the `Exchange` as an optional 2nd parameter. This is usable if the type converter for instance needs information from the current exchange. For instance combined with the encoding support its possible for type converters to convert with the configured encoding. An example from camel-core for the `byte[] -> String` converter:

```
@Converter
public static String toString(byte[] data, Exchange exchange) {
    if (exchange != null) {
        String charsetName = exchange.getProperty(Exchange.CHARSET_NA
        if (charsetName != null) {
            try {
                return new String(data, charsetName);
            } catch (UnsupportedEncodingException e) {
                LOG.warn("Can&#39;t convert the byte to String with th
charsetName, e);
            }
        }
    }
    return new String(data);
}
```