Creating a new Transport Handler

To create a new TransportHandler, the Interface ITransportHandler must be implemented or the class AbstractTransportHandler be extended.

Depending on the way, the handler works, different methods need to be implemented.

Independent of Push/Pull

```
public ITransportHandler createInstance(IProtocolHandler<?> protocolHandler, Map<String, String> options);
```

This method must return a new initialized transport handler. Typically, the constructor is called. When using Procedural Query Language (PQL) the options map is generated automatically from the option field (e.g. Access operator)

Example of FileHandler

```
    @Override
    public ITransportHandler createInstance(
            IProtocolHandler<?> protocolHandler, Map<String, String> options) {
        return new FileHandler(protocolHandler, options);
    }
```

The methode getName() must deliver a global unique transport handler name.

```
    String getName();
```

Its a good was to use this a follows (again example of FileHandler):

```
        public static final String NAME = "File";

        @Override
    public String getName() {
        return NAME;
    }
```

When implementing ITransportHandler, open and close need to be implemented.

Hint: In the following we will assume, that AbstractTransportHandler will be overwritten.

AbstractTransportHandler provides already default implementations that cannot be overwritten (its implementend in AbstractTransportHandlerDelegate):

```
    final synchronized public void open() throws UnknownHostException,
            IOException {
        if (openCounter == 0) {
            if (getExchangePattern() != null
                    && (getExchangePattern().equals(
                            ITransportExchangePattern.InOnly)
                            || getExchangePattern().equals(
                                    ITransportExchangePattern.InOptionalOut) || getExchangePattern()
                            .equals(ITransportExchangePattern.InOut))) {
                callOnMe.processInOpen();
            }
            if (getExchangePattern() != null
                    && (getExchangePattern().equals(
                            ITransportExchangePattern.OutOnly)
                            || getExchangePattern().equals(
                                    ITransportExchangePattern.OutOptionalIn) || getExchangePattern()
                            .equals(ITransportExchangePattern.InOut))) {
                callOnMe.processOutOpen();
            }
        }
        openCounter++;
    }

    final synchronized public void close() throws IOException {
        openCounter--;
        if (openCounter == 0) {
            if (getExchangePattern() != null
                    && (getExchangePattern().equals(
                            ITransportExchangePattern.InOnly)
                            || getExchangePattern().equals(
                                    ITransportExchangePattern.InOptionalOut) || getExchangePattern()
                            .equals(ITransportExchangePattern.InOut))) {
                callOnMe.processInClose();
            }
            if (getExchangePattern() != null
                    && (getExchangePattern().equals(
                            ITransportExchangePattern.OutOnly)
                            || getExchangePattern().equals(
                                    ITransportExchangePattern.OutOptionalIn) || getExchangePattern()
                            .equals(ITransportExchangePattern.InOut))) {
                callOnMe.processOutClose();
            }
        }
    }
```

A TransportHandler can provide different exchange pattern. The handler must deliver the pattern when calling the following method:

```
    public ITransportExchangePattern getExchangePattern();
```

Currently, the following values are available (https://en.wikipedia.org/wiki/Message_Exchange_Pattern):

- InOnly: The handler can only be used as source.
- RobustInOnly
- InOut: The handler can be used as source and as sink.
- InOptionalOut
- OutOnly: The handler can only be used as sink.
- RobustOutOnly
- OutIn
- OutOptionalIn

AbstractTransportHandler calls according to the exchange pattern the corresponding methods:

- processInOpen()
- processOutOpen()
- processInClose()
- processOutClose()

In this methods the TransportHandler must open or close the connections. The "IN"-methods are called for sources, the "OUT" for sinks. When starting or stopping a query, open and close are called respectively.

In the following again the implemenations for the FileHandler

```
    @Override
    public void processInOpen() throws IOException {
        if (!preload) {
            final File file = new File(filename);
            try {
                in = new FileInputStream(file);
                fireOnConnect();
            } catch (Exception e) {
                fireOnDisconnect();
                throw e;
            }
        } else {
            fis = new FileInputStream(filename);
            FileChannel channel = fis.getChannel();
            long size = channel.size();
            double x = size / (double) Integer.MAX_VALUE;
            int n = (int) Math.ceil(x);
            ByteBuffer buffers[] = new ByteBuffer[n];
            for (int i = 0; i < n; i++) {
                buffers[i] = ByteBuffer.allocateDirect(Integer.MAX_VALUE);
                channel.read(buffers[i]);
                buffers[i].rewind();
            }
            in = createInputStream(buffers);
            fireOnConnect();
        }
    }

    @Override
    public void processInClose() throws IOException {
        super.processInClose();
        if (fis != null) {
            fis.close();
        }
    }

    @Override
    public void processOutOpen() throws IOException {
        final File file = new File(filename);
        try {
            out = new FileOutputStream(file, append);
            fireOnConnect();
        } catch (Exception e) {
            fireOnDisconnect();
            throw e;
        }
    }

    @Override
    public void processOutClose() throws IOException {
        fireOnDisconnect();
        out.flush();
        out.close();
    }
```

Imporant: Use processInStart if the transport handler connects to a service that immediatly sends data!

Hint: There are scenarios in which it is not feasible to separate transport and protocol layer. In such cases, one can implement the combination as a transport handler and use it in combination with the "None" protocol handler.

Generic Pull

After the connection is inialized, the framework tries to retrieve data from the TransportHandler. To be generic, we decided to use an InputStream for sources and an OutputStream for sinks. So the following methods need to be overwritten (Remark: It its not necessary to implement both methods, if the TransportHandler e.g. should only be used for sources):

```
public InputStream getInputStream();
public OutputStream getOutputStream();
```

A typical implementation in FileHandler:

```
    @Override
    public InputStream getInputStream() {
        return in;
    }
    @Override
    public OutputStream getOutputStream() {
        return out;
    }
```

In some cases, sources deliver not an endless data stream. For such cases the method

```
 public boolean isDone();
```

can be overwrittten.

Simple Transport Handler

Another way, implementing a transport handler that is less generic and delivers e.g. tuples directly can be found in the next example. In this case, the protocol handler must be 'none' ..

```
package de.uniol.inf.is.odysseus.wrapper.temper1.physicaloperator.access;


import java.util.Map;
import java.util.Random;

import de.uniol.inf.is.odysseus.core.collection.Tuple;
import de.uniol.inf.is.odysseus.core.physicaloperator.access.protocol.IProtocolHandler;
import de.uniol.inf.is.odysseus.core.physicaloperator.access.transport.AbstractSimplePullTransportHandler;
import de.uniol.inf.is.odysseus.core.physicaloperator.access.transport.ITransportHandler;

public class RandomTransportHandler extends AbstractSimplePullTransportHandler<Tuple<?>>{

    private static final String NAME = "Random";
    private static final Random RAND = new Random();

    @Override
    public ITransportHandler createInstance(IProtocolHandler<?> protocolHandler, Map<String, String> options) {
        RandomTransportHandler tHandler = new RandomTransportHandler();

        protocolHandler.setTransportHandler(tHandler);

        return tHandler;
    }

    @Override
    public String getName() {
        return NAME;
    }

    private static float readDevice() {
        return 20f + ( 10 * RAND.nextFloat());
    }

    @Override
    public boolean hasNext() {
        return true;
    }

    @SuppressWarnings("rawtypes")
    @Override
    public Tuple<?> getNext() {
        Tuple<?> tuple = new Tuple(1, false);
        tuple.setAttribute(0, readDevice());
        return tuple;
    }

    @Override
    public boolean isSemanticallyEqualImpl(ITransportHandler other) {
        return false;
    }

}
```

GenericPush

In generic push szenarios for sources there is no method that can be overwritten because it depends on the transport type and e.g. libararies that receive data from external sources. The information that is read must be send to the corresponding transport handler. To simplify the process, AbstractTransportHandler(Delegate) provides the following methods that should be used:

```
    public void fireProcess(ByteBuffer message) {
        for (ITransportHandlerListener<T> l : transportHandlerListener) {
            // TODO: flip() erases the contents of the message if
            // it was already flipped or just created...
            // In other words: This method expects that the byte buffer
            // is not fully prepared
            message.flip();
            l.process(message);
        }
    }

    public void fireProcess(T m) {
        for (ITransportHandlerListener<T> l : transportHandlerListener) {
            l.process(m);
        }
    }
    public void fireProcess(String[] message) {
        for (ITransportHandlerListener<T> l : transportHandlerListener) {
            l.process(message);
        }
    }
```

The fireProcess methods can be used with ByteBuffers and String-Arrays or with a Generic. In the latter case, the corresponding ProtocolHandler must read this type, else a class cast exception will be thrown.

Important: A transport handler must not send data before processInStart is called!

An example to the use fireProcess-Methods (and processInOpen/processInStart) can be found in the RabbitMQ transport handler:

```
    @Override
public void processInOpen() throws IOException {
    try {
        internalOpen();
    } catch (TimeoutException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
}

@Override
public void processInStart() {
    try {
        if (publishStyle == PublishStyle.PublishSubscribe) {
            String queueName = channel.queueDeclare().getQueue();
            channel.queueBind(queueName, exchangeName, "");
        }

        // Create Consumer
        boolean autoAck = false;
        channel.basicConsume(queueName, autoAck, consumerTag, new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, com.rabbitmq.client.Envelope envelope,
                    com.rabbitmq.client.AMQP.BasicProperties properties, byte[] body) throws IOException {
                // String routingKey = envelope.getRoutingKey();
                // String contentType = properties.getContentType();
                long deliveryTag = envelope.getDeliveryTag();
                try {
                    ByteBuffer wrapped = ByteBuffer.wrap(body);
                    wrapped.position(wrapped.limit());
                    fireProcess(wrapped);
                } catch (Exception e) {
                    LOG.warn("Error processing input", e);
                }
                channel.basicAck(deliveryTag, false);
            };
        });

        connection.addShutdownListener(new ShutdownListener() {

            @Override
            public void shutdownCompleted(ShutdownSignalException cause) {
                LOG.warn("Connection shutdown.", cause);
            }
        });

        channel.addShutdownListener(new ShutdownListener() {

            @Override
            public void shutdownCompleted(ShutdownSignalException cause) {
                LOG.warn("Channel shutdown.", cause);
            }
        });
    } catch (IOException e) {
        throw new StartFailedException(e);
    }

}
```

Here you can see, that every source type needs a special handling for sending. Here e.g. a callback object (DefaultConsumer) is defined in RabbitMQ that calls fireProcess.

Two additional methods are used to inform listener about connection states (connect and disconnect)

```
    public void fireOnConnect(ITransportHandler handler) {
        for (ITransportHandlerListener<T> l : transportHandlerListener) {
            l.onConnect(handler);
        }
    }
    public void fireOnDisconnect(ITransportHandler handler) {
        for (ITransportHandlerListener<T> l : transportHandlerListener) {
            l.onDisonnect(handler);
        }
    }
```

These methode are defined in the interface ITransportHandlerListener that is implemented by IProtocolHandler, the basic interface for ProtocolHandler.

Registering the handler

Odysseus is OSGi based and all the handler are implemented as declartive services.

For this, you have to create a xml file, typically placed under a folder called OSGI-INF, where you state the global unique name of the handler, the implemenation class and the interface that this handler provides. In the following is the example for the FacebookTransportHandler.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="de.uniol.inf.is.odysseus.wrapper.facebook.
physicaloperator.access.FacebookTransportHandler">
   <implementation class="de.uniol.inf.is.odysseus.wrapper.facebook.physicaloperator.access.
FacebookTransportHandler"/>
   <service>
      <provide interface="de.uniol.inf.is.odysseus.core.physicaloperator.access.transport.ITransportHandler"/>
   </service>
</scr:component>
```

The file MANIFEST.MF (typically provided in META-INF) must contain a hint to this new file

```
Service-Component: OSGI-INF/FacebookTransportHandler.xml, OSGI-INF/FacebookProtocolHandler.xml
```

Remark: An eclipse wizzard can be used to create this file and the reference inside the MANIFEST.MF: File/New/Component Definition