

Access Operators

Extending the framework

AccessAO, AccessAOBuilder ↻ PQL Documentation
GenericPush and GenericPull
New Wrapper

Transport Handler

A Transport Handler is responsible for the communication between an arbitrary source or sink and Odysseus. Multiple Transport Handlers already exists in Odysseus and are documented [here](#). They also serve as a good starting point for own Transport Handler.

Push

To implement a Push Transport Handler (a transport handler that receives data instead of requests) an *AbstractPushTransportHandler* has to be extended. Only a few methods have to be implemented like the *send* method that transfers data to source or target of the transport, the *getName* method which returns the unique name of this transport handler, and the *create instance* method which creates a new instance of this transport handler with the given options. Further, ones has to implement the *process open* and *process close* methods. These methods are used to open or close the transport channel depending on the exchange pattern of the transport handler. There are 8 possible transport pattern based on the Message Exchange Pattern (MEP) namely they are: InOnly, RobustInOnly, InOut, InOptionalOut, OutOnly, RobustOutOnly, OutIn, and OutOptionalIn.

Example Push Transport Handler

```
public class ExampleTransportHandler extends AbstractPushTransportHandler {
    /** Logger */
    private Logger LOG = LoggerFactory.getLogger(ExampleTransportHandler.class);

    public ExampleTransportHandler() {
        super();
    }
    public ExampleTransportHandler(IProtocolHandler<?> protocolHandler) {
        super(protocolHandler);
    }
    @Override
    public void send(byte[] message) throws IOException {

    }
    @Override
    public ITransportHandler createInstance(
        IProtocolHandler<?> protocolHandler, Map<String, String> options) {
        ExampleTransportHandler handler = new ExampleTransportHandler(
            protocolHandler);
        // Set options
        return handler;
    }

    @Override
    public String getName() {
        return "Example";
    }
    @Override
    public void processInOpen() throws IOException {

    }
    @Override
    public void processOutOpen() throws IOException {

    }
    @Override
    public void processInClose() throws IOException {

    }
    @Override
    public void processOutClose() throws IOException {

    }
    @Override
    public ITransportExchangePattern getExchangePattern() {
        return ITransportExchangePattern.InOnly;
    }
}
```

Pull

The implementation of a Pull Transport Handler is similar to the Push Transport Handler except the fact that ones has to implement the two methods `getInputStream` and `getOutputStream` that are used by a protocol handler to request the next data element.

Example Pull Transport Handler

```
public class ExampleTransportHandler extends AbstractPullTransportHandler {
    /** Logger */
    private Logger LOG = LoggerFactory.getLogger(ExampleTransportHandler.class);
    private InputStream in;
    private OutputStream out;
    public ExampleTransportHandler() {
        super();
    }
    public ExampleTransportHandler(IProtocolHandler<?> protocolHandler) {
        super(protocolHandler);
    }
    @Override
    public void send(byte[] message) throws IOException {

    }
    @Override
    public ITransportHandler createInstance(
        IProtocolHandler<?> protocolHandler, Map<String, String> options) {
        SpeechTransportHandler handler = new SpeechTransportHandler(
            protocolHandler);
        // Set options
        return handler;
    }

    @Override
    public String getName() {
        return "Example";
    }
    @Override
    public void processInOpen() throws IOException {

    }
    @Override
    public void processOutOpen() throws IOException {

    }
    @Override
    public void processInClose() throws IOException {

    }
    @Override
    public void processOutClose() throws IOException {

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

Protocol Handler

A Protocol Handler is responsible for the conversion between an arbitrary data model and the Odysseus data model. Multiple Protocol Handlers already exists in Odysseus and are documented [here](#). They also serve as a good starting point for own Protocol Handler.

To implement a Protocol Handler an *AbstractProtocolHandler* has to be extended. A few method has to be implemented here. First of all the method *getName* has to be implemented that returns the unique name of the Protocol Handler. After that, the method *createInstance* need to be implemented to create a new instance of the Protocol Handler with the given options. By the fact, that a Protocol Handler can be used in a pull and a push fashion, the methods *hasNext* and *getNext* for pull access and *process* for push access have both to be implemented to read data using the data handler and forward the data to the transfer. In addition, the method *write* has to be implemented to write data. Further, the methods *open* and *close* to open and close the Transport Handler and initialize or shutdown the the source/sink should be implemented. Other methods like the *onConnect* and *onDisconnect* can be implemented to treat events in the transport channel adequately. Depending on the applications ones want to override the methods *getExchangePattern* to define the supported exchange pattern for the transport direction (int or out) and access (push or pull).

Protocol Handler

```
public class ExampleProtocolHandler<T> extends AbstractProtocolHandler<T> {

    public ExampleProtocolHandler() {
        super();
    }
    public ExampleProtocolHandler(ITransportDirection direction,
        IAccessPattern access) {
        super(direction, access);
    }
    @Override
    public void open() throws UnknownHostException, IOException {
        getTransportHandler().open();
    }
    @Override
    public void close() throws IOException {
        getTransportHandler().close();
    }
    @Override
    public boolean hasNext() throws IOException {
        return true;
    }
    @Override
    public T getNext() throws IOException {
        return getDataHandler().readData("data");
    }
    @Override
    public void write(T object) throws IOException {

    }
    @Override
    public IPProtocolHandler<T> createInstance(ITransportDirection direction,
        IAccessPattern access, Map<String, String> options,
        IDataHandler<T> dataHandler, ITransferHandler<T> transfer) {
        ExampleProtocolHandler<T> instance = new ExampleProtocolHandler<T>(direction,
            access);
        instance.setDataHandler(dataHandler);
        instance.setTransfer(transfer);
        // Set options
        return instance;
    }
    @Override
    public String getName() {
        return "Example";
    }
    @Override
    public void onConnect(ITransportHandler caller) {

    }
    @Override
    public void onDisonnect(ITransportHandler caller) {

    }
    @Override
    public void process(ByteBuffer message) {
        getTransfer().transfer(getDataHandler().readData(message));
    }
}
```

