clearpoint

BUSINESS DIRECTION
PROCESS CLARITY
TECHNOLOGY DESIGN

Mule Demo

Mule

Walk Through

Mule Overview

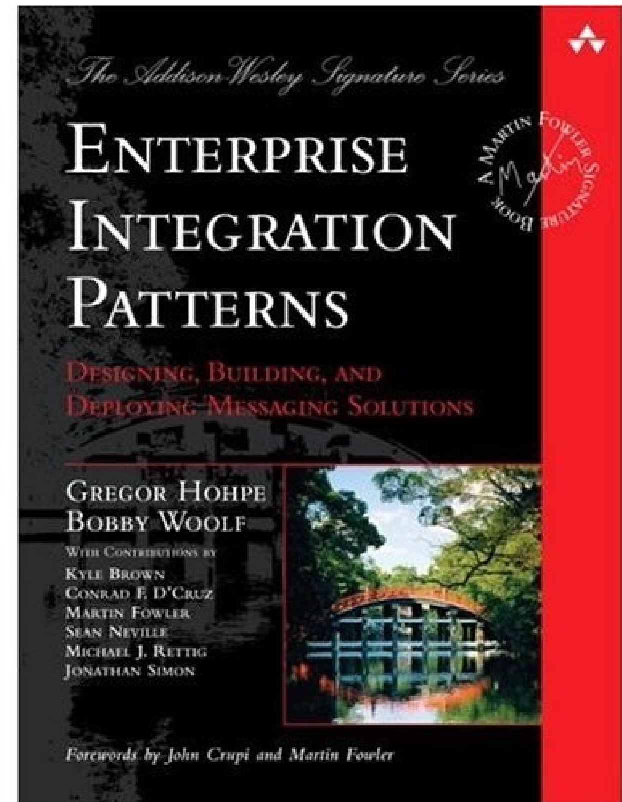
- Mule is used in the tech demo to glue the Orderman, Stockcontrol and Dashboard applications together
- Mule is a Java based ESB
- Mule provides over 30 transports out of the box including: email, web services, file, database, messaging and FTP

Mule Overview Cont.

- Mule runs in multiple topologies and can be clustered for high availability and processing through-put
- Mule uses a “staged event-driven architecture” (SEDA) architecture which improves modularity and ensures good behaviour under high-load

EIP & Mule

- Mule uses patterns and terminology from the Enterprise Integration Patterns book by Hohpe & Woolf



Key Mule Concepts

Messages

- At the simplest level Mule provides a framework for reading, transforming and sending data between applications in the form of messages
- A message may be anything: plain text, binary data, XML, a serialised Java bean etc.

Service Components

- Service components perform business logic on messages
- Service components are managed by Mule but don't have any Mule-specific code
- Service components can be Plain Old Java Objects (POJO), Spring beans, Java beans or web services

Inbound Router

- An Inbound Router controls which messages are delivered to a Service Component for processing
- They can filter, aggregate and resequence messages before passing them to the Service Component for processing

Outbound Router

- An Outbound Router takes the output of a Service Component (in the form of messages) and dispatches them for processing by other components
- Outbound routers can be chained and can have filters to control where messages are dispatched to

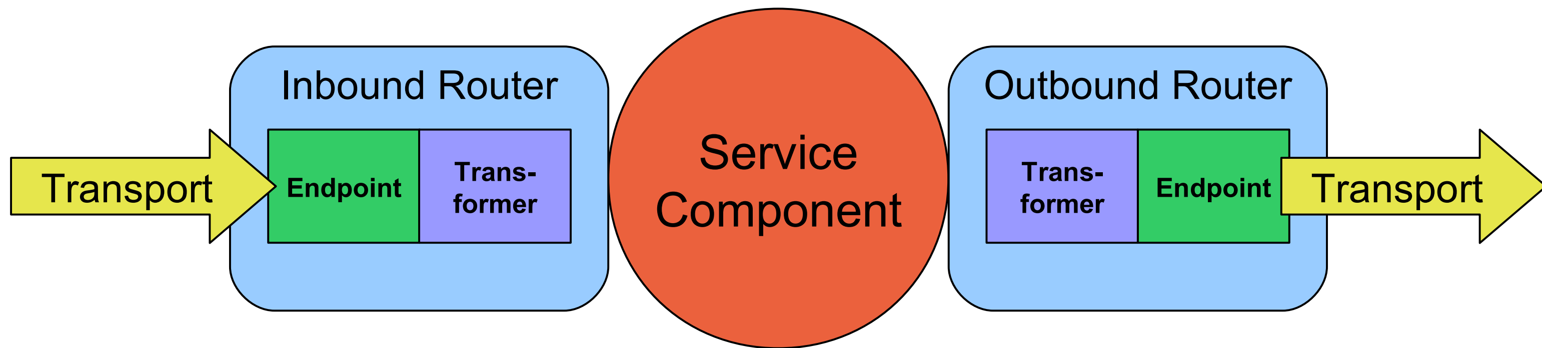
Transports & Transformers

- Transports implement logic to route messages over different protocols such as HTTP, JMS, email etc.
- A Transformer transforms a message into a format that can be processed by a Service Component

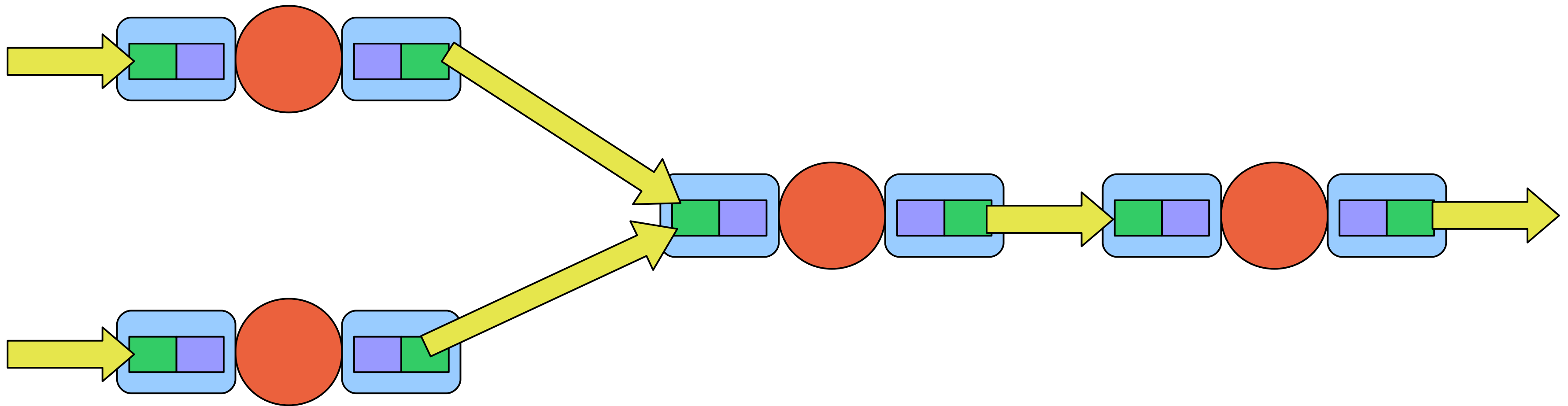
Endpoints

- Endpoints are used to wire together services
- The are specified on Inbound and Outbound routers
- They tell Mule which transport to use, where to send messages, and which messages a service component should receive

Putting it together...



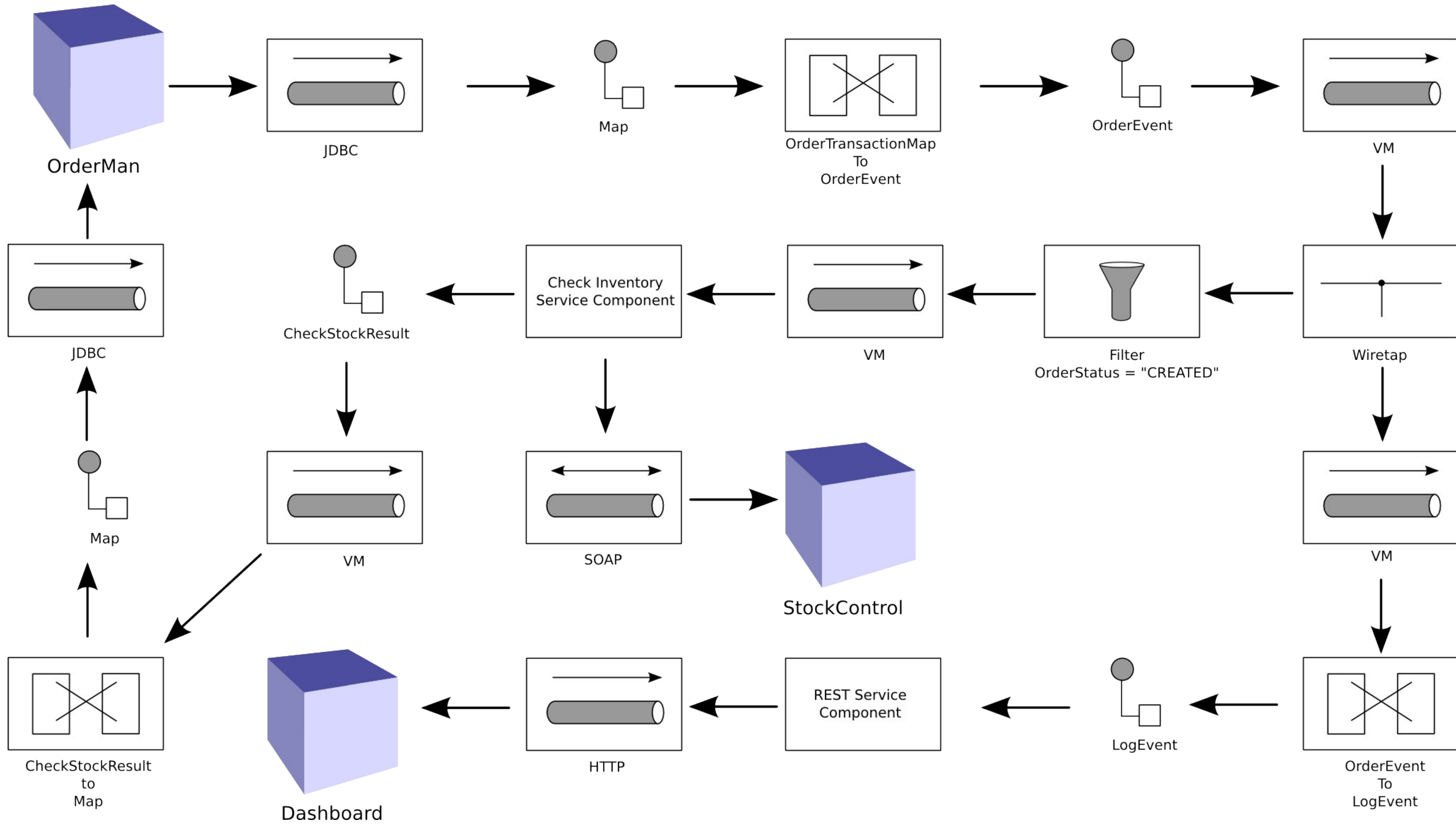
Putting it together



clearpoint

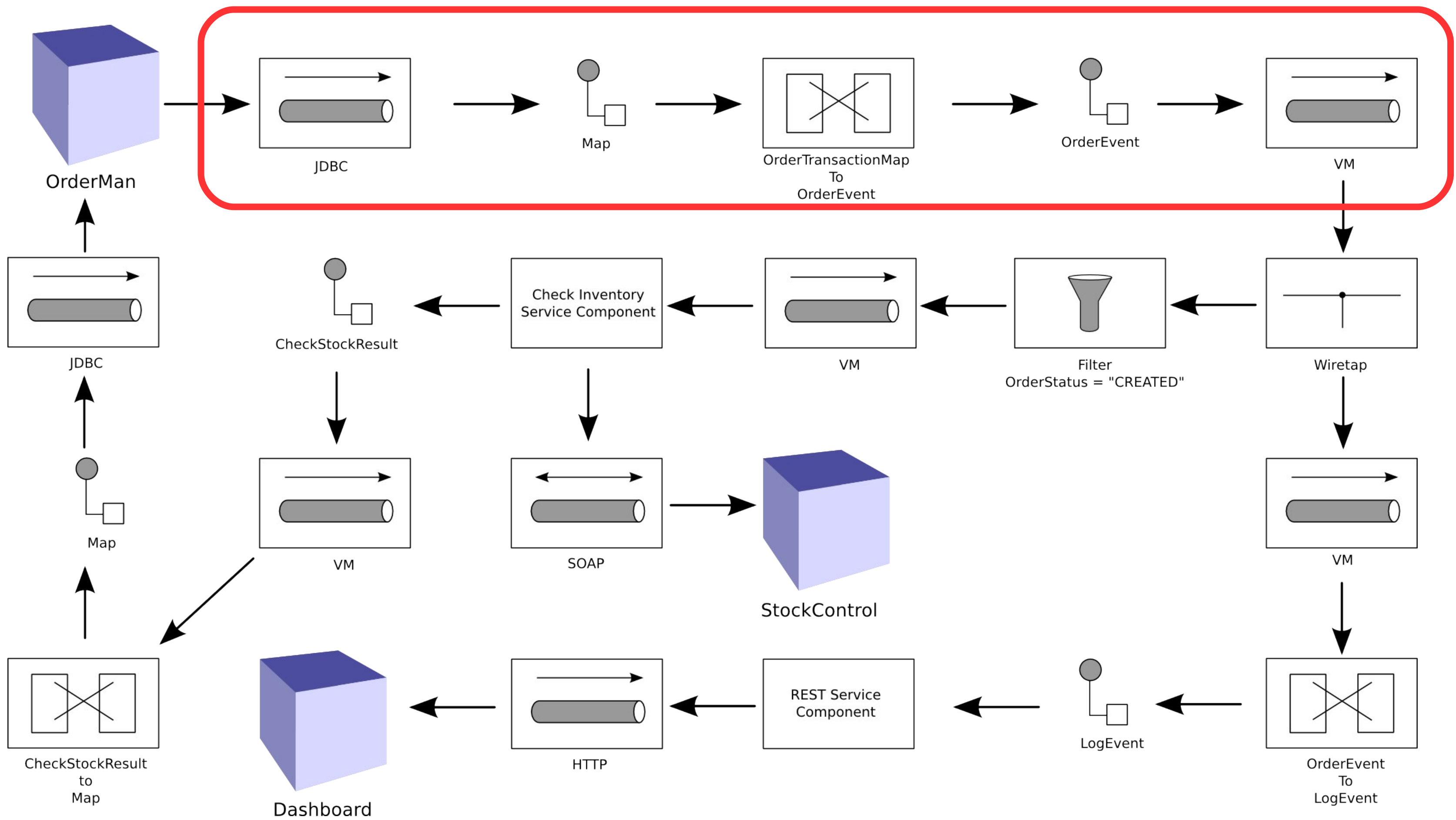
Solution Walk Through

Solution Overview



clearpoint

Step 1: Extract Order Events



clearpoint

OrderTransaction Table & Stored Procedures

```
CREATE TABLE IF NOT EXISTS `OrderTransaction` (
  `TransID` int(11) NOT NULL auto_increment,
  `OrderId` int(11) NOT NULL,
  `TransDateTime` timestamp NOT NULL default CURRENT_TIMESTAMP,
  `OrderStatus` varchar(10) NOT NULL,
  PRIMARY KEY  (`TransID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

CREATE TRIGGER `aqorderman`.`AIOrder` AFTER INSERT ON `aqorderman`.`Order`
 FOR EACH ROW BEGIN
 INSERT INTO OrderTransaction(OrderId,OrderStatus)
  VALUES (NEW.OrderID,NEW.OrderStatus);
END

CREATE TRIGGER `aqorderman`.`AUOrder` AFTER UPDATE ON `aqorderman`.`Order`
 FOR EACH ROW BEGIN
 INSERT INTO OrderTransaction(OrderId,OrderStatus)
  VALUES (NEW.OrderID,NEW.OrderStatus);
END
```

The Datasource

```
<spring:bean id="OrderManDatasource"
            class="org.enhydra.jdbc.standard.StandardDataSource"
      destroy-method="shutdown">
            <spring:property name="driverName" value="com.mysql.jdbc.Driver" />
            <spring:property name="url"
                  value="jdbc:mysql://127.0.0.1/aqorderman" />
            <spring:property name="user" value="aqorderman" />
            <spring:property name="password" value="password" />
</spring:bean>
```

The JDBC connector

```
<jdbc:connector name="orderManJdbcConnector"
            pollingFrequency="10000" dataSource-ref="OrderManDatasource">

  <jdbc:query key="selectOrderTransactions"
      value="SELECT * FROM OrderTransaction ot,`Order` o WHERE
        ot.OrderId = o.OrderId ORDER BY TransId" />

  <jdbc:query key="selectOrderTransactions.ack"
      value="DELETE FROM OrderTransaction WHERE
        TransId = #[map-payload:TransId]" />

. . .
```

The ExtractOrderEvents Service

```
<service name="ExtractOrderEvents">
  <inbound>
    <jdbc:inbound-endpoint queryKey="selectOrderTransactions"
         connector-ref="orderManJdbcConnector"
     transformer-refs="OrderTransactionMapToOrderEvent" />
  </inbound>
  <outbound>
    <pass-through-router>
          <outbound-endpoint ref="orderEventsQueue" />
        </pass-through-router>
  </outbound>
</service>
```

The Transformer

- The JDBC connector returns a Hashmap (name-value pairs)
- Mule doesn't force any particular message format
- Going to use a Java bean/POJO
- The OrderTransactionMapToOrderEvent transformer handles the conversion

The OrderEvent Bean

```
public class OrderEvent implements Serializable
{
. . .

  /** The order Id. */
  private Integer orderId;
  /** The order date time. */
  private Calendar orderDateTime;
  /** The customer's email. */
  private String customerEmail;

. . .

  public Integer getOrderId()
  {
    return orderId;
  }

. . .
```

OrderTransactionMapToOrderEvent

```
public class OrderTransactionMapToOrderEvent extends AbstractTransformer
{
  /**
   * Constructor. Registers the source and result types.
   *
   */
  public OrderTransactionMapToOrderEvent()
  {
    super();
    this.registerSourceType(Map.class);
    this.setReturnClass(OrderEvent.class);
  }

. . .
```

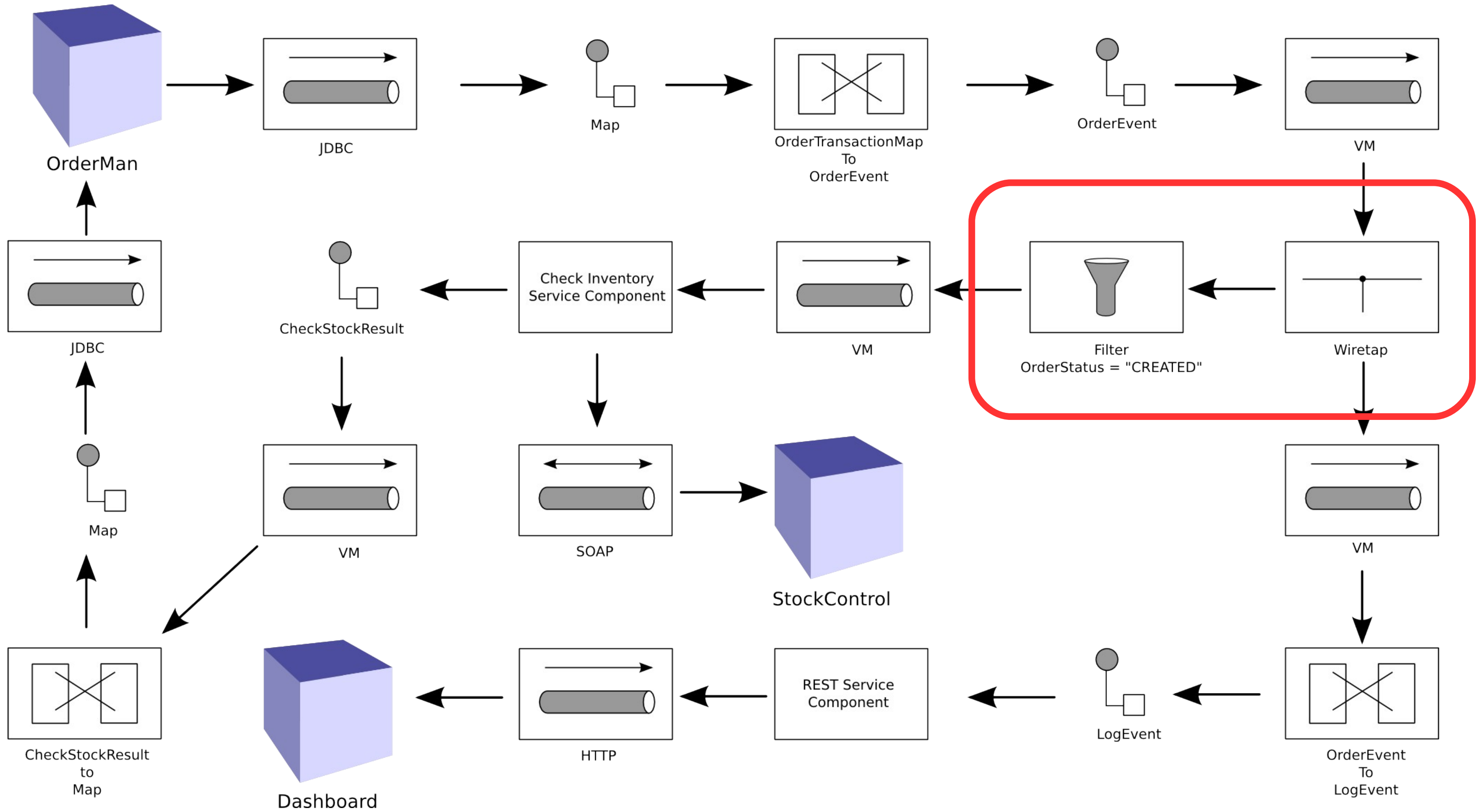
OrderTransactionMapToOrderEvent cont.

```
  public Object doTransform(Object src, String encoding)
     throws TransformerException
  {
    // get the source as a Map
    Map sourceMap = (Map) src;
. . .
    // create a new OrderEvent
    OrderEvent tempEvent = new OrderEvent();
    // populate values from map
    tempEvent.setCustomerEmail((String) sourceMap.get("customeremail"));
. . .
    tempEvent.setEventDateTime(tempCal);
. . .
    tempEvent.setOrderId((Integer) sourceMap.get("orderid"));
    tempEvent.setStatus((String) sourceMap.get("orderstatus"));
. . .
    // return the event
    return tempEvent;
  }
```

Transformer & Endpoint Defs

```
<custom-transformer name="OrderTransactionMapToOrderEvent"
 class="nz.co.clearpoint.aqdemo.mule.transformer.
       OrderTransactionMapToOrderEvent" />


<endpoint name="orderEventsQueue" address="vm://orderevents" />
```

Step 2: Process Order Events



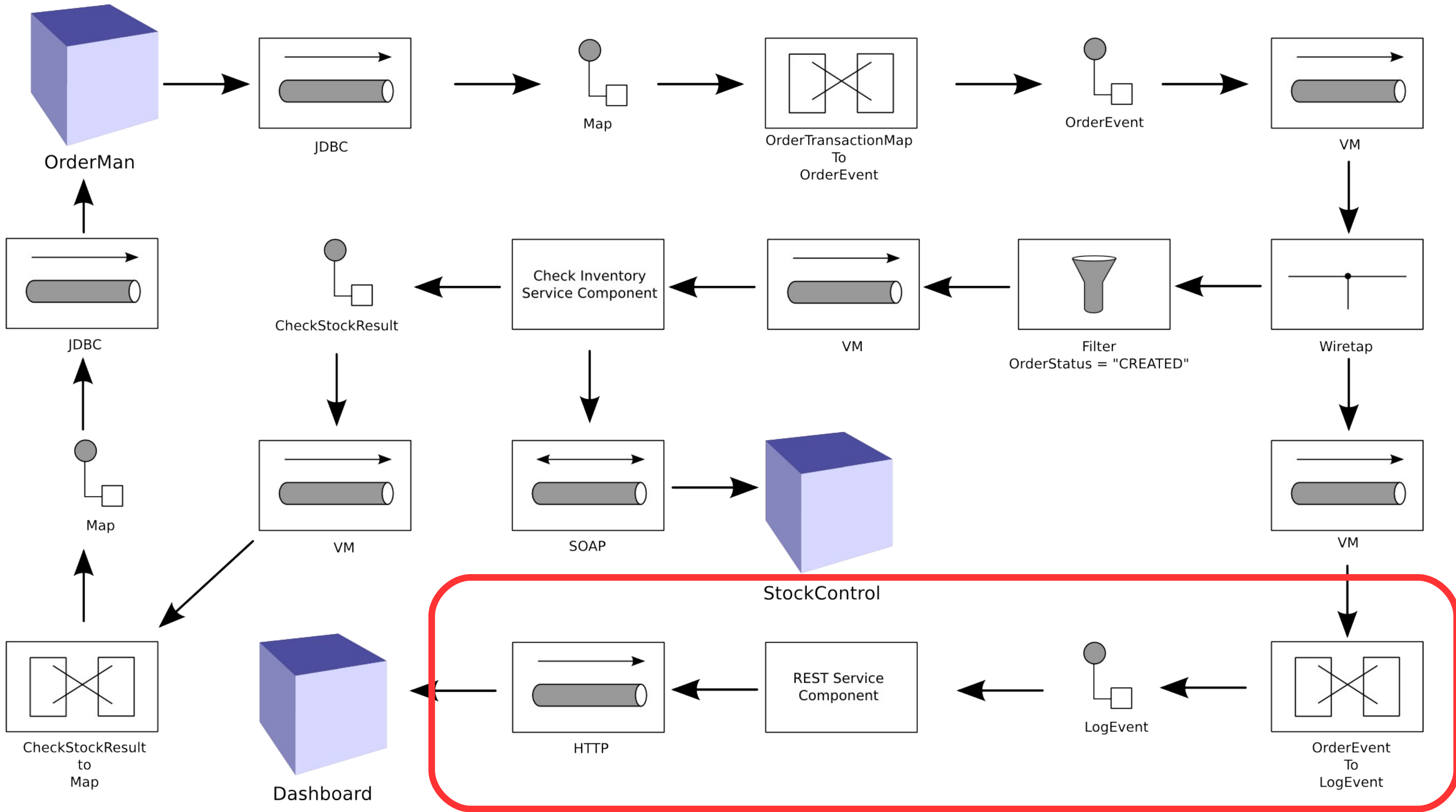
clearpoint

The ProcessOrderEvents Service

```
<service name="ProcessOrderEvents">
        <inbound>
                <inbound-endpoint ref="orderEventsQueue" />
                <wire-tap-router>
                        <vm:outbound-endpoint ref="logOrderEventQueue" />
                </wire-tap-router>
        </inbound>
        <outbound>
                <filtering-router>
                        <outbound-endpoint ref="checkInventoryQueue" />
                        <expression-filter evaluator="jxpath"
                    expression="status='CREATED'" />
                </filtering-router>
            <!-- just log order events we don't handle -->
                <logging-catch-all-strategy />
        </outbound>
</service>
```

clearpoint

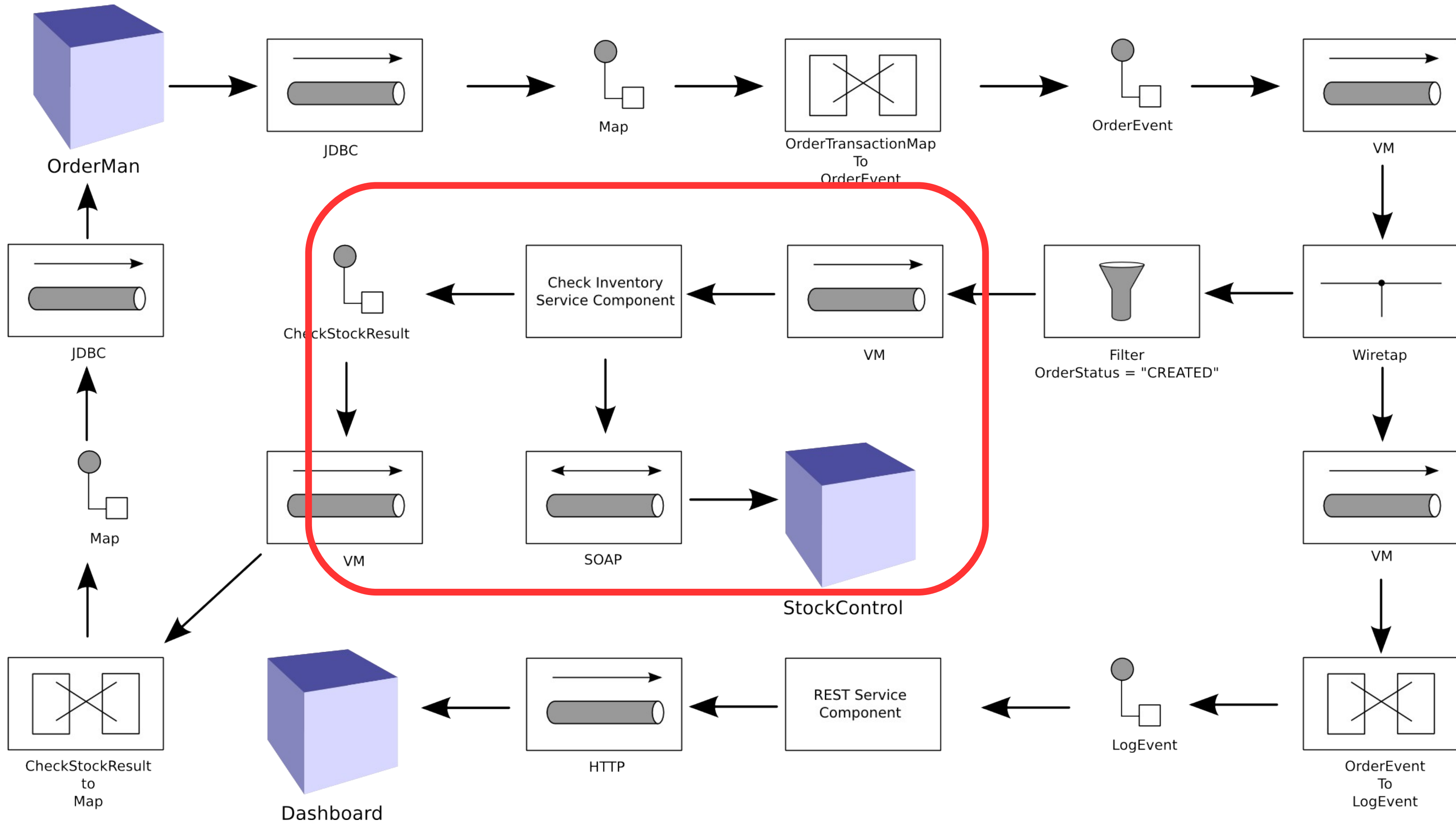
Step 3: Log Order Events



The LogOrderEvents Service

```
<service name="LogOrderEvents">
	<inbound>
		<inbound-endpoint ref="logOrderEventQueue"
			transformer-refs="OrderEventToLogEvent" />
	</inbound>
	<http:rest-service-component
		serviceUrl="http://localhost:8080/dashboard/LogEvent.action"
		httpMethod="GET">
		<http:error-filter>
			<wildcard-filter pattern="*FAIL*" />
		</http:error-filter>
		<http:requiredParameter key="objectId"
			value="#[bean:objectId]" />
		<http:requiredParameter key="eventTypeId"
			value="#[bean:eventTypeId]" />
. . .
	</http:rest-service-component>
</service>
```

Step 4: Check Inventory



clearpoint

The CheckInventory Service

```
<service name="CheckInventory">
    <inbound>
        <inbound-endpoint ref="checkInventoryQueue" />
    </inbound>
    <component class="nz.co.clearpoint.aqdemo.mule.component.CheckInventory">
. . .
    </component>
    <outbound>
        <pass-through-router>
            <outbound-endpoint ref="checkStockResultQueue" />
        </pass-through-router>
    </outbound>
</service>
```

The CheckInventory Component

- This component needs to call the stock control web service and then merge the response with data from the original OrderEvent message

The CheckInventory Component Cont.

```
public class CheckInventory
{
. . .
  public Object checkInventory(OrderEvent orderEvent)
  {
    // call the check stock service to see if everything is in stock
    boolean serviceResult = checkStockService.checkStock(
                        orderEvent.getWidget1Count(),
                        orderEvent.getWidget2Count(),
                        orderEvent.getWidget3Count());
    // create a new result object
    CheckStockResult result = new CheckStockResult();
    // populate the result
    result.setOrderId(orderEvent.getOrderId());
    result.setAllStockAvailable(serviceResult);
    // return the result
    return result;
  }
}
```

clearpoint

Binding component

- The binding component feature of Mule allows a component to be decoupled from the details of calling a service (in this case the CheckStock web service)

The CheckStockService Interface

- This interface defines the service that will be called
- The CheckInventory component simply uses the class supplied by Mule that implements this interface
- The CheckInventory component has no visibility of how the service is implemented

The CheckStock Interface

```
public interface CheckStockService
{
  boolean checkStock(int widget1, int widget2, int widget3);
}
```

The CheckInventory Component Cont.

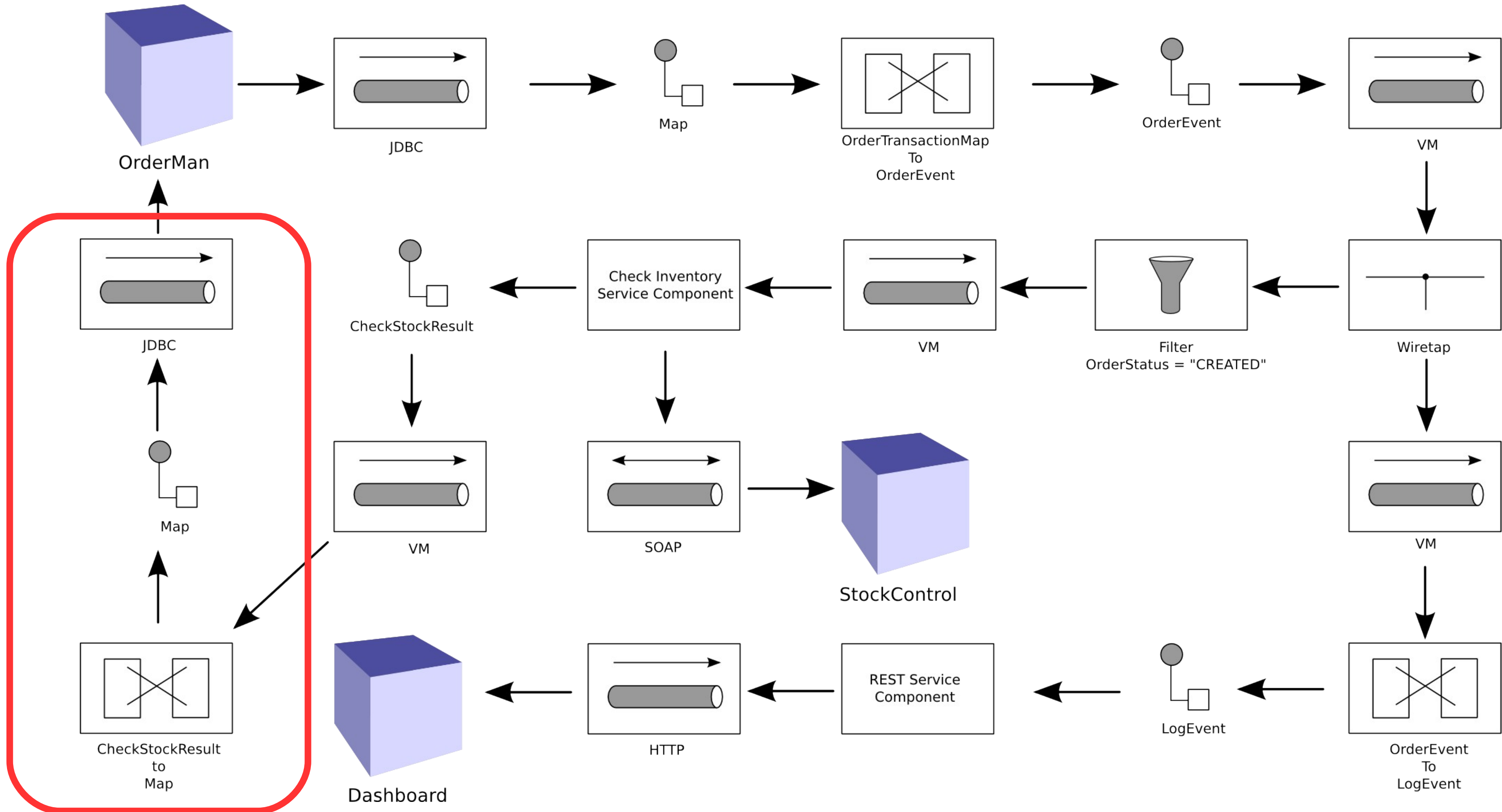
```
. . .
  /** Ref for CheckStockService binding component. */
  private CheckStockService checkStockService;

  public CheckStockService getCheckStockService()
  {
    return checkStockService;
  }

  public void setCheckStockService(CheckStockService checkStockService)
  {
    this.checkStockService = checkStockService;
  }

. . .
```

Binding to the service

```
<binding interface="nz.co.clearpoint.aqdemo.mule.
        component.CheckStockService">
    <outbound-endpoint
    address="wsdl-cxf:http://127.0.0.1:8080/stockcontrol/services/
  CheckStockService1x0?WSDL&amp;method=checkStock" />
</binding>
```

- This section defines the binding for the CheckStockService
- The wsdl-cxf binding automatically generates a SOAP client from the service's WSDL

Step 5: Update OrderMan With Stock Check Result



clearpoint

The UpdateOrderMan WithStockCheckResult Srv.

```
<service name="UpdateOrderManWithStockCheckResult">
        <inbound>
                <inbound-endpoint ref="checkStockResultQueue"
                        transformer-refs="CheckStatusResultToMap" />
        </inbound>

        <outbound>
                <pass-through-router>
                        <jdbc:outbound-endpoint
                 queryKey="updateWithStockCheckResults"
                                connector-ref="orderManJdbcConnector" />
                </pass-through-router>
        </outbound>
</service>
```

The JDBC query

```
<jdbc:connector name="orderManJdbcConnector"
            pollingFrequency="10000" dataSource-ref="OrderManDatasource">
. . .
  <jdbc:query key="updateWithStockCheckResults"
                   value="UPDATE `Order` SET
               OrderStatus = #[map-payload:orderStatus] WHERE
               OrderId = #[map-payload:orderId]
               AND OrderStatus != 'CANCELLED'" />
</jdbc:connector>
```

clearpoint