Internals of IBM Integration Bus

David Coles
IBM Integration Bus Level 3 Technical Lead,
IBM Hursley – dcoles@uk.ibm.com

7th August 2014
16200

#SHAREorg

SHARE is an independent volunteer-run information technology association that provides **education**, professional **networking** and industry **influence**.

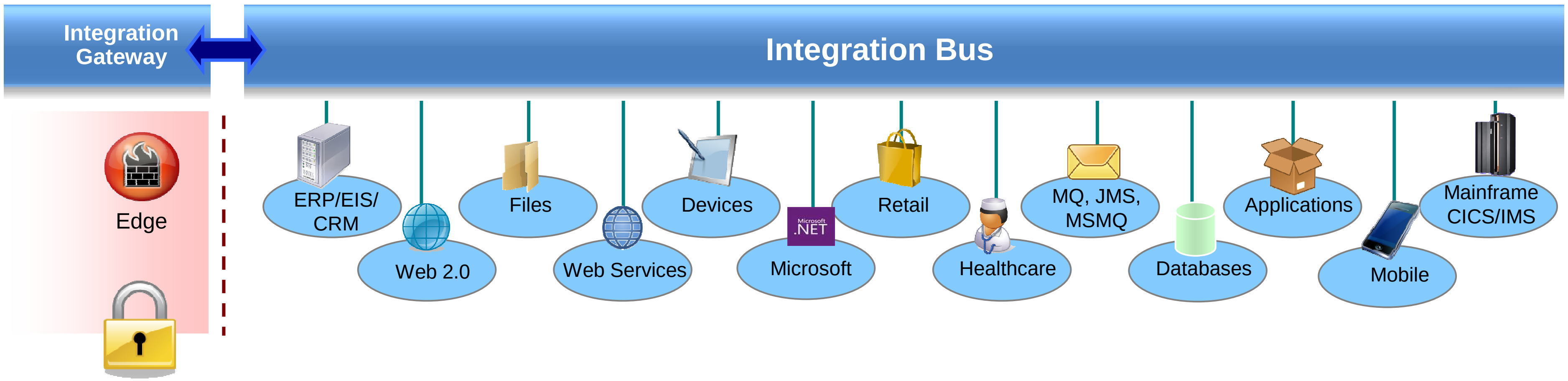
SHARE in Pittsburgh 2014

Agenda

- Introduction
- Runtime Processes
- Threads
- Memory
- Diagnostic Information
- External Internals
  - Execution Engine / Stack Size
  - Node lifecycle
  - Parsers and the Logical Tree

Introducing IBM Integration Bus

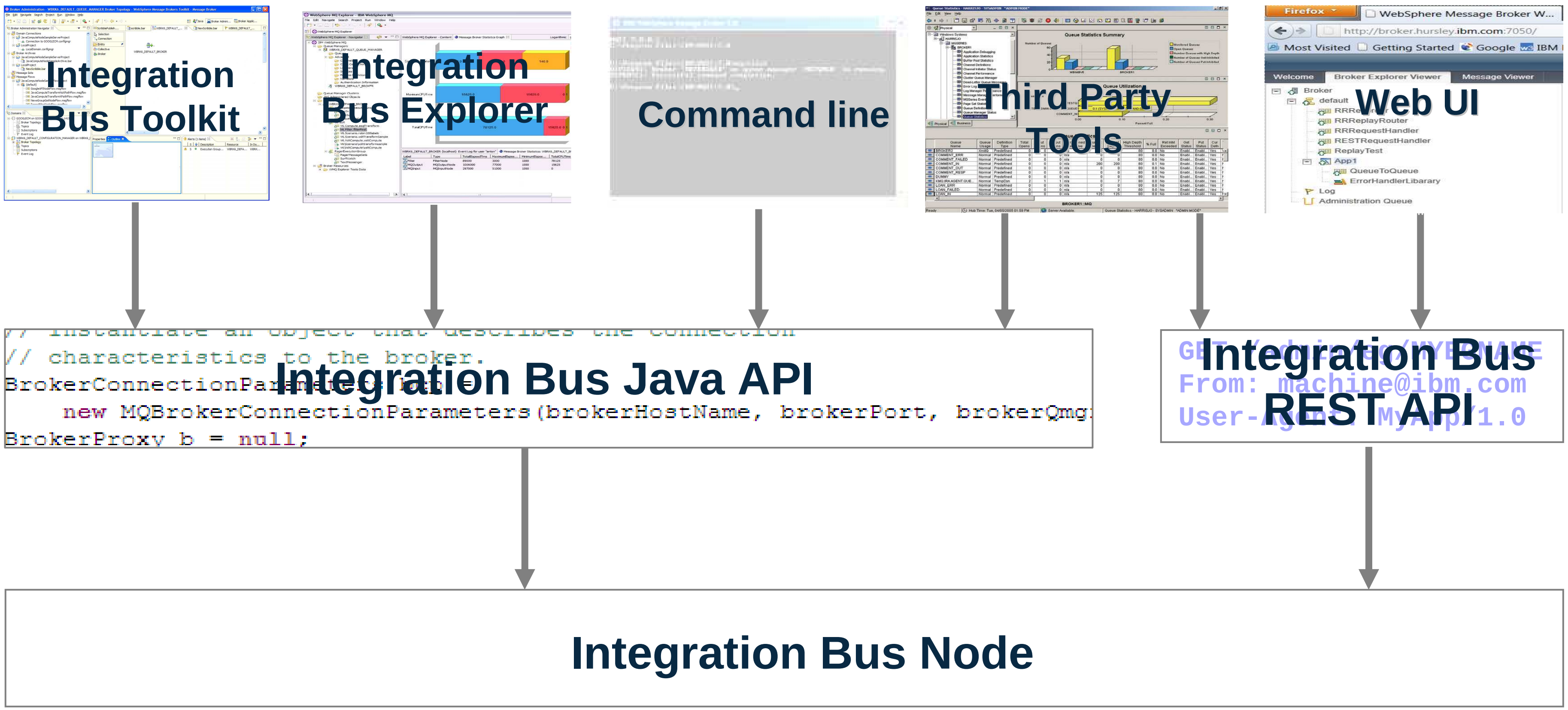
- **IBM's Strategic Integration Technology**
  - Single engineered product for .NET, Java and fully heterogeneous integration scenarios
  - DataPower continues to evolve as IBM's integration gateway



- **A Natural Evolution for WebSphere Message Broker users**
  - Significant innovation and evolution of WMB technology base
  - New features for Policy-based WLM, BPM integration, Business rules and .NET

- **Designed to incorporate WebSphere Enterprise Service Bus use cases**
  - Capabilities for WESB are folded in to IBM Integration Bus over time
  - Conversion tools for initial use cases built in to IIB from day one
  - WESB technology remains in market, supported. Migrate to Integration Bus when ready

Integration Bus Components



Runtime Processes

Processes

- **bipimain**
  - z/OS only. The first process in any IIB address space. APF authorised to set up authorised routines.
- **bipservice**
  - Lightweight and resilient process that starts and monitors the bipbroker process (a.k.a AdminAgent).
  - If the bipbroker process fails, bipservice will restart it.
- **bipbroker**
  - A more substantial process. Contains the deployment manager, for CMP and REST connections, as well as the WebUI server. All commands, toolkit connections and WebUI go through this process.
  - Responsible for starting and monitoring the biphttplistener and DataFlowEngine processes.
  - If either process fails, bipbroker will restart them.
- **biphttplistener**
  - Runs the brokerwide HTTP connector for HTTP and SOAP nodes.
- **DataFlowEngine** (a.k.a Integration Server)
  - Runtime engine for all deployed resources.

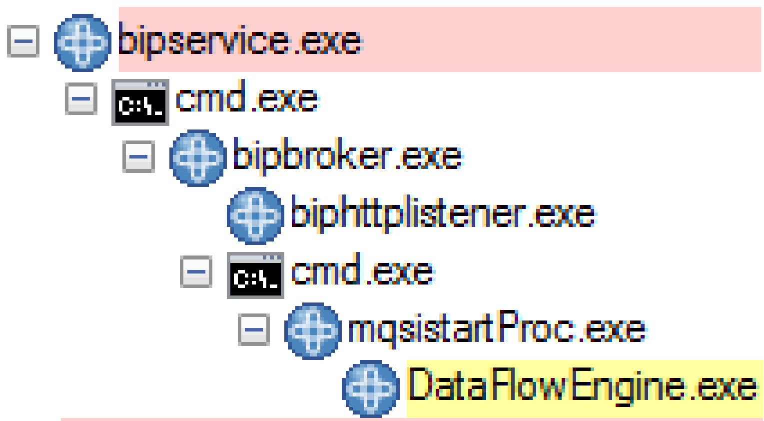
Processes by platform

- Linux and Unix systems

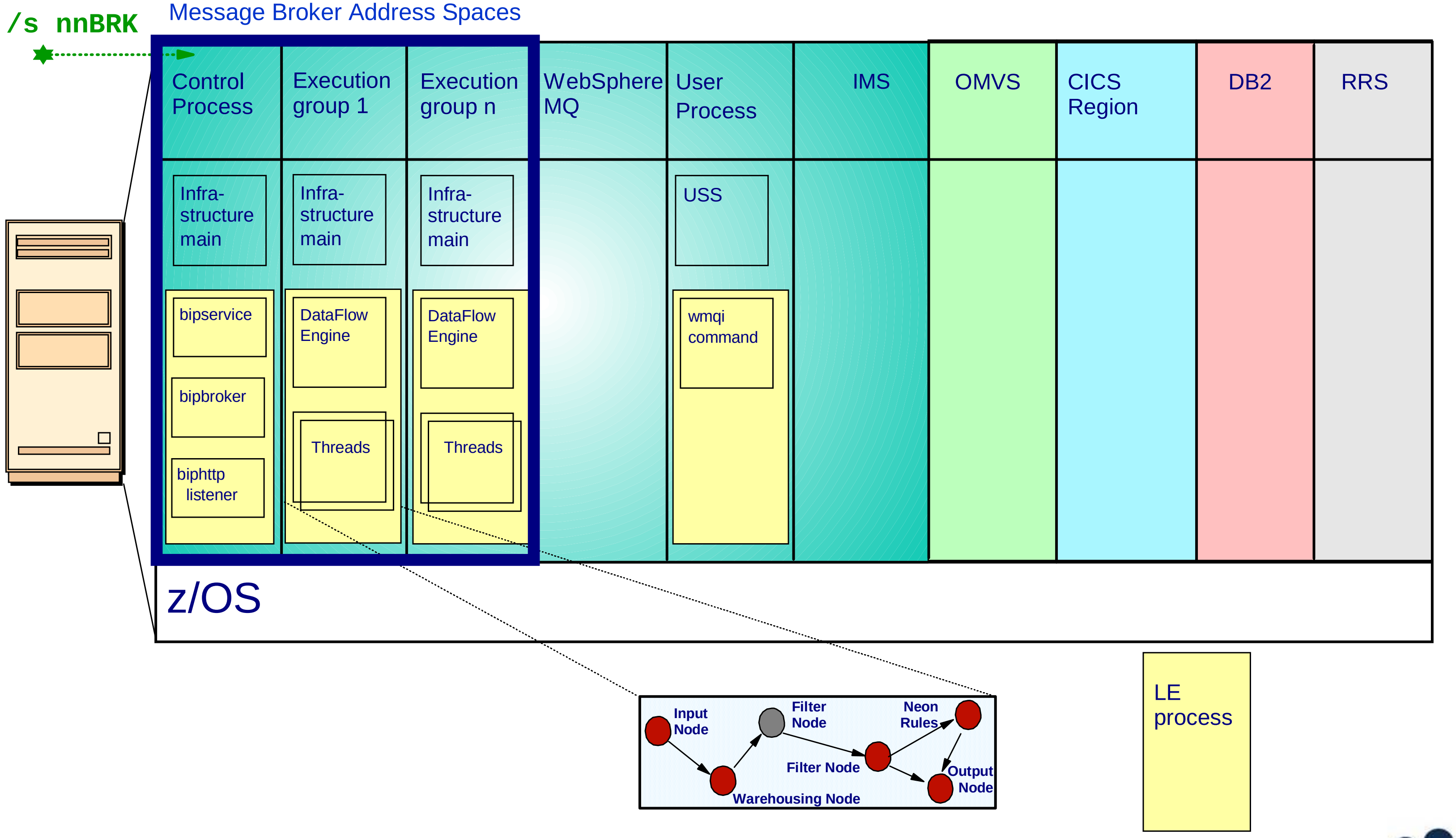
```
        PID      ParentPID
mqm     25475         1  0 15:56 ?        00:00:00 bipservice CSIM
mqm     25480 25475  0 15:56 ?        00:00:26 bipbroker CSIM
mqm     25556 25480  0 15:56 ?        00:00:01 biphttplistener CSIM

mqm       722 25480  0 16:28 ?        00:00:00 /bin/bash /opt/mqsi/bin/startDataFlowEngine
mqm       723   722  0 16:28 ?        00:00:10 DataFlowEngine CSIM d8ffc588-4401-0000-0080
```

- Windows



Processes grouped by Address Space on z/OS



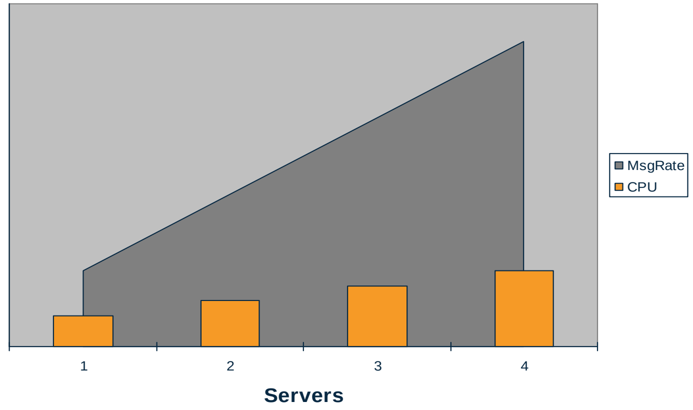
Displaying broker at the process level on z/OS

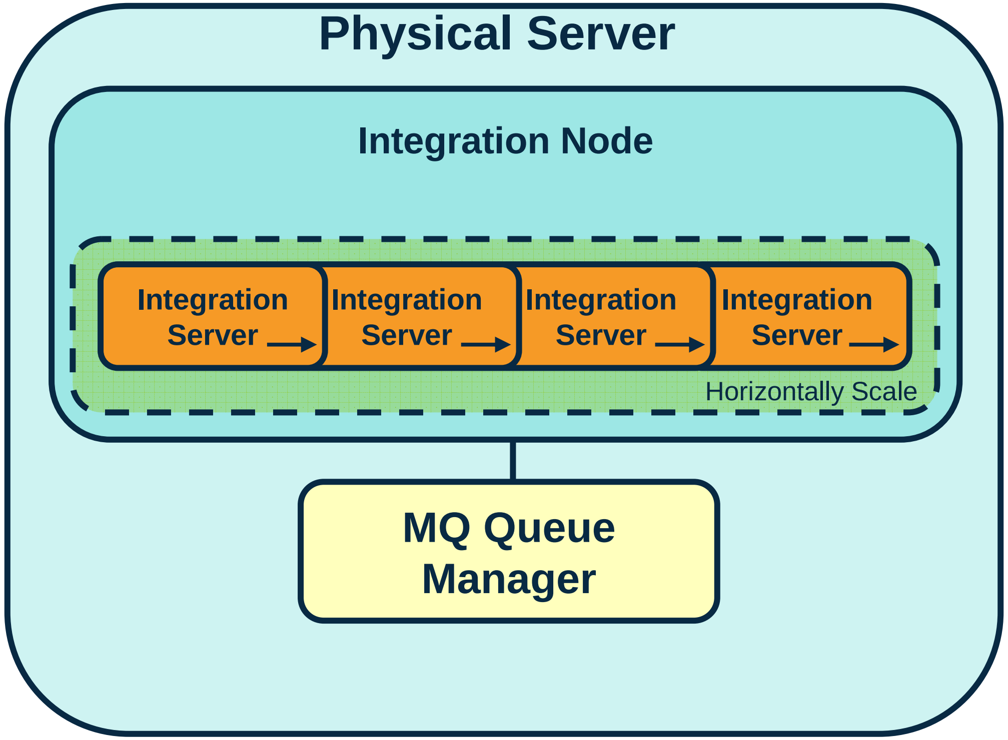
```
SDSF PROCESS DISPLAY  MVD1     ALL                        LINE 1-11 (11)
COMMAND INPUT ===> _                                           SCROLL ===> CSR
PREFIX=MQ05BRK  DEST=(ALL)  OWNER=*  SORT=ASID/A  SYSNAME=*
NP    JOBNAME   JobID    Status                         Owner    Command
      MQ05BRK   STC52908 FILE SYS KERNEL WAIT           MQ05BRK  BPXBATA8
      MQ05BRK   STC52908 RUNNING                        MQ05BRK  biphttplistener MQ05BRK
      MQ05BRK   STC52908 RUNNING                        MQ05BRK  bipservice MQ05BRK AUTO
      MQ05BRK   STC52908 WAITING FOR CHILD              MQ05BRK  /argoinst/DAVE/usr/lpp/mqsi/
      MQ05BRK   STC52908 RUNNING                        MQ05BRK  bipbroker MQ05BRK
      MQ05BRK   STC52910 RUNNING                        MQ05BRK  DataFlowEngine MQ05BRK 4faaf
      MQ05BRK   STC52910 WAITING FOR CHILD              MQ05BRK  /argoinst/DAVE/usr/lpp/mqsi/
      MQ05BRK   STC52910 FILE SYS KERNEL WAIT           MQ05BRK  BPXBATA8
      MQ05BRK   STC52909 RUNNING                        MQ05BRK  DataFlowEngine MQ05BRK cc32d
      MQ05BRK   STC52909 FILE SYS KERNEL WAIT           MQ05BRK  BPXBATA8
      MQ05BRK   STC52909 WAITING FOR CHILD              MQ05BRK  /argoinst/DAVE/usr/lpp/mqsi/
```

```
/u/gormand:>ps -ef | grep MQ05
 MQ05BRK  84017209  50463369 /argoinst/DAVE/usr/lpp/mqsi/bin/bipimain bipservice MQ05BRK AUTO
 MQ05BRK  84017284  84017835 DataFlowEngine MQ05BRK cc32dfb6-3001-0000-0080-a253e63dfe32 DAVE
 MQ05BRK  67240352  84017924 biphttplistener MQ05BRK
 MQ05BRK  67240502  84017209 bipservice MQ05BRK AUTO
 MQ05BRK  84017835  84017546 /argoinst/DAVE/usr/lpp/mqsi/bin/bipimain DataFlowEngine 00071016
     DFS    131792    131828 grep MQ05
 MQ05BRK  84017924  67240502 bipbroker MQ05BRK
```

Horizontal Scaling with additional processes

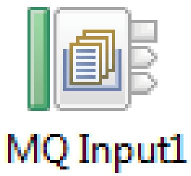
- IBM Integration Bus supports horizontal scaling. This is achieved by increasing the number of integration servers the service is running in.
- Reasons to add additional integration servers:
  - Increased throughput
  - Operational simplicity
  - Workload isolation
  - Better H/W utilisation
  - Higher Availability





Threads

Integration Flow threads (instances)

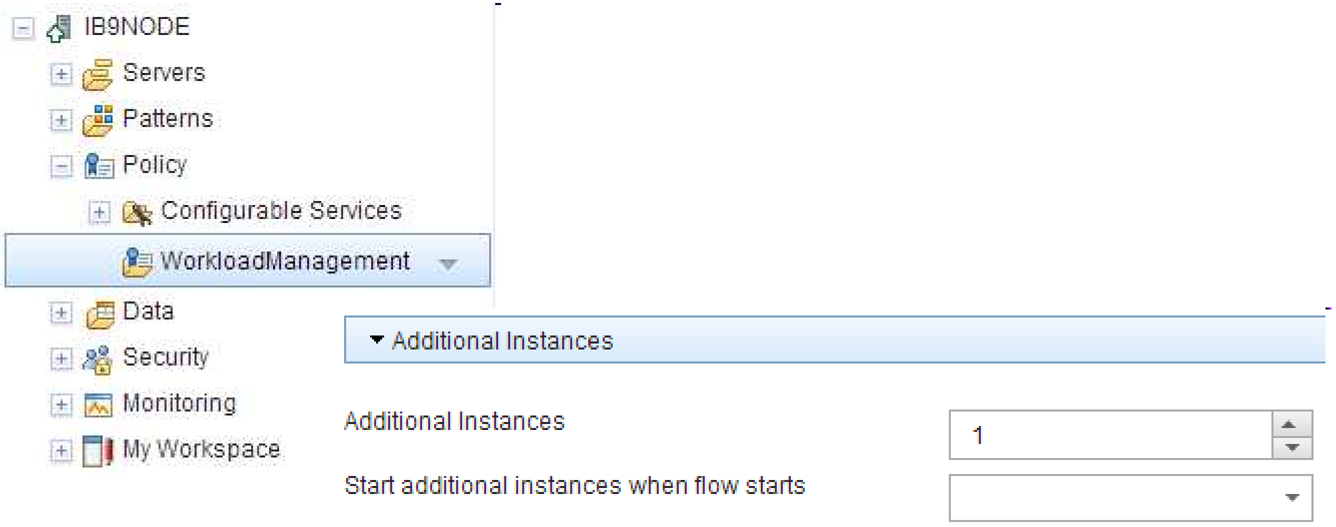


- Every flow will have at least 1 thread per input node.
  - On z/OS, these threads are TCBs.
- Held within a pool for each input node.
- Increase threads by adding “additional instances”.
- The integration server will start additional instances on demand, as the amount of incoming work increases, up to the limit specified.
- If they are idle then additional instances are stopped.
- Additional instances <u>can</u> be started immediately when the integration server starts.

- Be aware of adding additional instance at the flow level when there are multiple input nodes in the same flow!

Setting additional instances

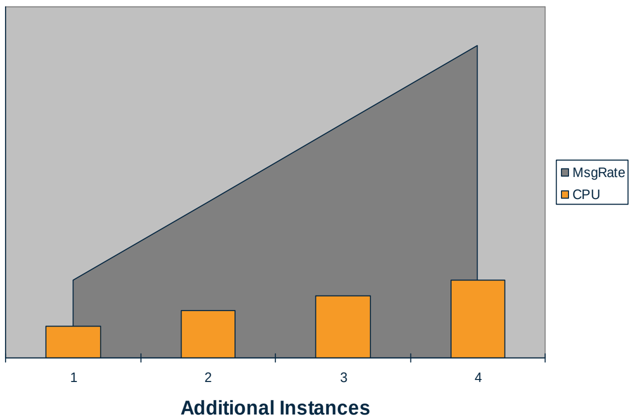
- In IIB V9, additional instances can be set by creating a workload management policy in the WebUI

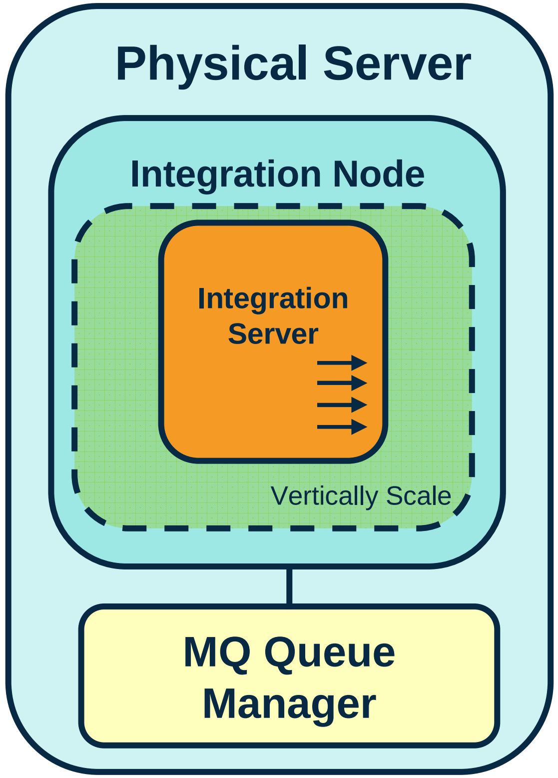


- Additional instances can also be defined on the input node, as a BAR file override and from IIB Explorer.

Vertical Scaling with additional threads

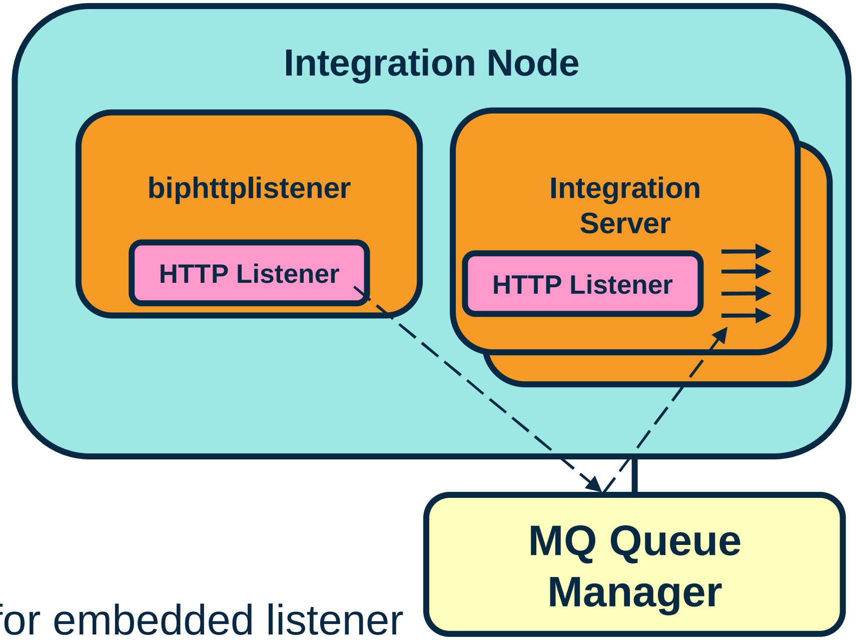
- IBM Integration Bus supports vertical scaling. This is achieved by increasing additional instances (threads) of the service running within the integration server.
- Reasons to add additional instances:
  - Increased throughput
  - Lower memory requirement
  - Better H/W utilisation
  - Policy management





Other thread pools

- There are other thread pools for the HTTP connectors.



- HTTP Connector for embedded listener
  - mqsireportproperties IB9NODE -e default -o HTTPConnector -a
- HTTP Connector for broker wide listener
  - mqsireportproperties IB9NODE -b httplistener -o HTTPConnector -a

HTTP Connector parameters

```
HTTPConnector
  uuid='HTTPConnector'
  address=''
  port='7080'
  maxPostSize=''
  acceptCount=''
  compressableMimeTypes=''
  compression=''
  connectionLinger=''
  connectionTimeout=''
  maxHttpHeaderSize=''
  maxKeepAliveRequests=''
  maxThreads=''
  minSpareThreads=''
  noCompressionUserAgents=''
  restrictedUserAgents=''
  socketBuffer=''
  tcpNoDelay=''
  enableLookups='false'
```

- Check the IIB Infocenter for a list of all the HTTP Connector parameters, and their default values.

-n maxThreads

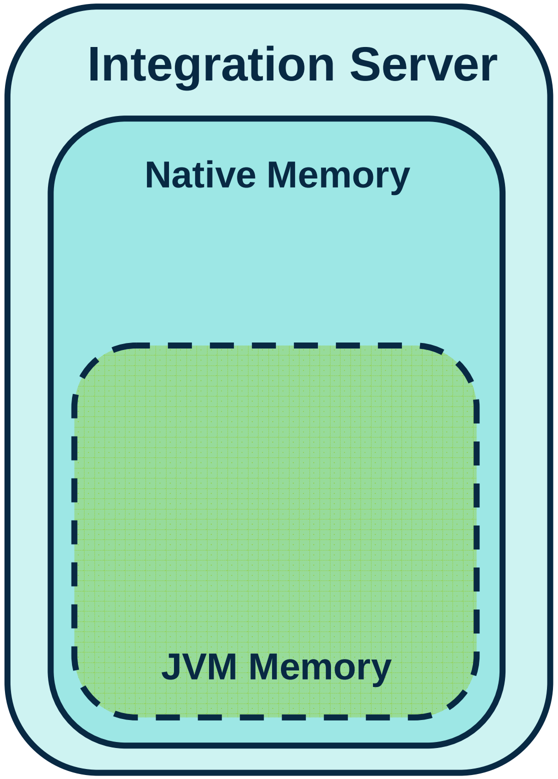
Set the value to the maximum number of threads that can be created by the HTTPConnector.

- Value type - integer
- Initial value - 200

Memory

Memory use within an Integration Server

- The Integration Server runtime comprises both C++ and Java. This means there is a multi faceted approach to checking memory.



- The Integration Server is a process (DataFlowEngine). Its size is limited by available memory and configuration on the operating system.

- There is 1 JVM within each Integration Server. The JVM has preconfigured minimum and maximum HEAP settings.

What are the main consumers of memory?

- Normal (low) consumers of memory include:
  - Deployment artefacts
  - Configuration, such as increasing the min JVM HEAP
  - Message flows
  - Additional instance
  - …
- What to watch for:
  - Size of input messages (1K or 1GB)
  - How messages are parsed (whole file or records)
  - Message Tree copying (transformation nodes)
  - Custom code (ESQL, Java etc)
  - Caching (Shared Variable, GlobalCache etc)
  - JVM settings

See Parser Stats later!

How to check process memory usage

- Use operating system tools to view the DataFlowEngine process size
  - Windows. Process Explorer

Process	PID	C...	Private Bytes
DataFlowEngine.exe	13196	0.02	265,524 K

  - AIX

```
-bash-3.00$ ps -e -o vsz=,rss=,comm= | grep Data
190456 190496 DataFlowEngine
```

  - Unix/Linux

```
-bash-3.2$ ps -e -o vsz,rss,cmd | grep DataFlowEngine
 57984  1400 /bin/bash /opt/mqsi/bin/startDataFlowEngine CSIM bba82c62-7344-44b3-9568-1790fdab483b default
1137476 200208 DataFlowEngine CSIM bba82c62-7344-44b3-9568-1790fdab483b default
```

  - z/OS

NP	JOBNAME	StepName	ProcStep	JobID	Owner	SrvClass	RptClass	SysName	C	Pos	DP	Real	Paging	SIO	CPU%	ASID	ASIDX	EX
	MQ05BRK	LMEXPERT	BROKER	STC63060	MQ05BRK	STCFAST	COLIN	MVD1		IN	F4	55,031	0.00	3.62	0.03	205	00CD	
	MQ05BRK	MQ05BRK	BROKER	STC63059	MQ05BRK	STCUSER	MQ05	MVD1		IN	EC	30,901	0.00	0.00	0.05	208	00D0	
	MQ05BRK	DAVE	BROKER	STC63061	MQ05BRK	STCUSER	DQ05BASE	MVD1		IN	EC	56,046	0.00	3.62	0.03	240	00F0	

  - vsz = The size in kilobytes of the core image of the process
  - rss = Indicates the real memory (resident set) size of the process (in 1 KB units)
  - Real = 4k pages

How to check JVM memory usage

1. Check defaults:
   – mqsireportproperties IB9NODE -e default -o ComIbmJVMManager –a

```
jvmMinHeapSize='-1'
jvmMaxHeapSize='-1'
```

   - **jvmMinHeapSize**
     – *Initial value: -1, which represents 33554432 bytes (32MB) with the global cache disabled, or 100663296 (96MB) with the global cache enabled*
   - **jvmMaxHeapSize**
     – *Initial value: -1, which represents 268435456 bytes (256 MB)*

2. Check usage with Resource Statistics

default Resources Statistics (Snapshot time 12:19:47 - 12:20:07)

DotNet App Domains | CICS | DotNet GC | CORBA | ConnectDirect | DecisionServices | FTEAgent | FTP | File | GlobalCache | JDBCConnectionPools | JMS | JVM | ODBC | Parsers | SOAPInput | Sec

name	InitialMemoryInMB	UsedMemoryInMB	CommittedMemoryInMB	MaxMemoryInMB	CumulativeGCTimeInSeconds	CumulativeNumberOfGCCollections
summary	32	89	117	-1	4	102
Heap Memory	32	12	36	256		
Non-Heap Memory	0	77	81	-1		
Garbage Collection - Copy					0	44
Garbage Collection - MarkSweepCompact					4	58

Diagnostic Information

Diagnostic Information in WMB

- Diagnostic Information
  - Resource Statistics
  - Flow Statistics
  - Activity Log
  - Administration Log
  - System Log
  - Trace
  - Stdout/Stderr

Resource Statistics

- **Graphical Performance Monitor**
  - Reports comprehensive usage according of well known resources
    - Message Flows, Nodes, JVM, HTTP, SOAP/HTTP sockets etc
    - Optionally partitioned by Broker, Execution Group and Message Flow

- **Reporting Mechanisms**
  - Graphically reported through IIB Explorer
    - Sort, filter and chart performance characteristics
    - View CPU, IO and other metrics
    - Log data to file in CSV/Excel readable format for post processing
  - User Configurable Reporting Interval
    - XML report messages consumed by any end user application

- **Examples of Available Resource Report Metrics**
  - **JVM:** Memory used, thread count, heap statistics…
  - **Sockets:** Socket host/port open; bytes sent, bytes received

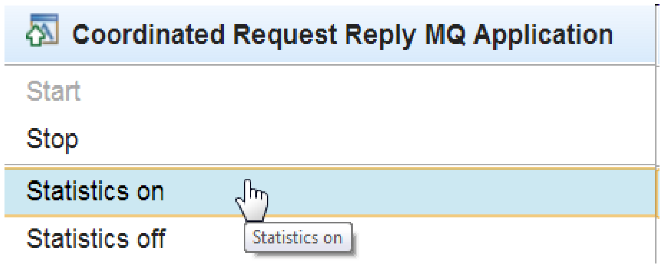
default Resources Statistics (Snapshot time 12:19:47 - 12:20:07)

DotNet App Domains | CICS | DotNet GC | CORBA | ConnectDirect | DecisionServices | FTEAgent | FTP | File | GlobalCache | JDBCConnectionPools | JMS | JVM | ODBC | Parsers | SOAPInput | Sec

name	InitialMemoryInMB	UsedMemoryInMB	CommittedMemoryInMB	MaxMemoryInMB	CumulativeGCTimeInSeconds	CumulativeNumberOfGCCollections
summary	32	89	117	-1	4	102
Heap Memory	32	12	36	256		
Non-Heap Memory	0	77	81	-1		
Garbage Collection - Copy					0	44
Garbage Collection - MarkSweepCompact					4	58

Flow Statistics

- Using the WebUI in Integration Bus v9:
  - Control statistics at all levels
  - Easily view and compare flows, helping to understand which are processing the most messages or have the highest elapsed time
  - Easily view and compare nodes, helping to understand which have the highest CPU or elapsed times.
  - View all statistics metrics available for each flow
  - View historical flow data

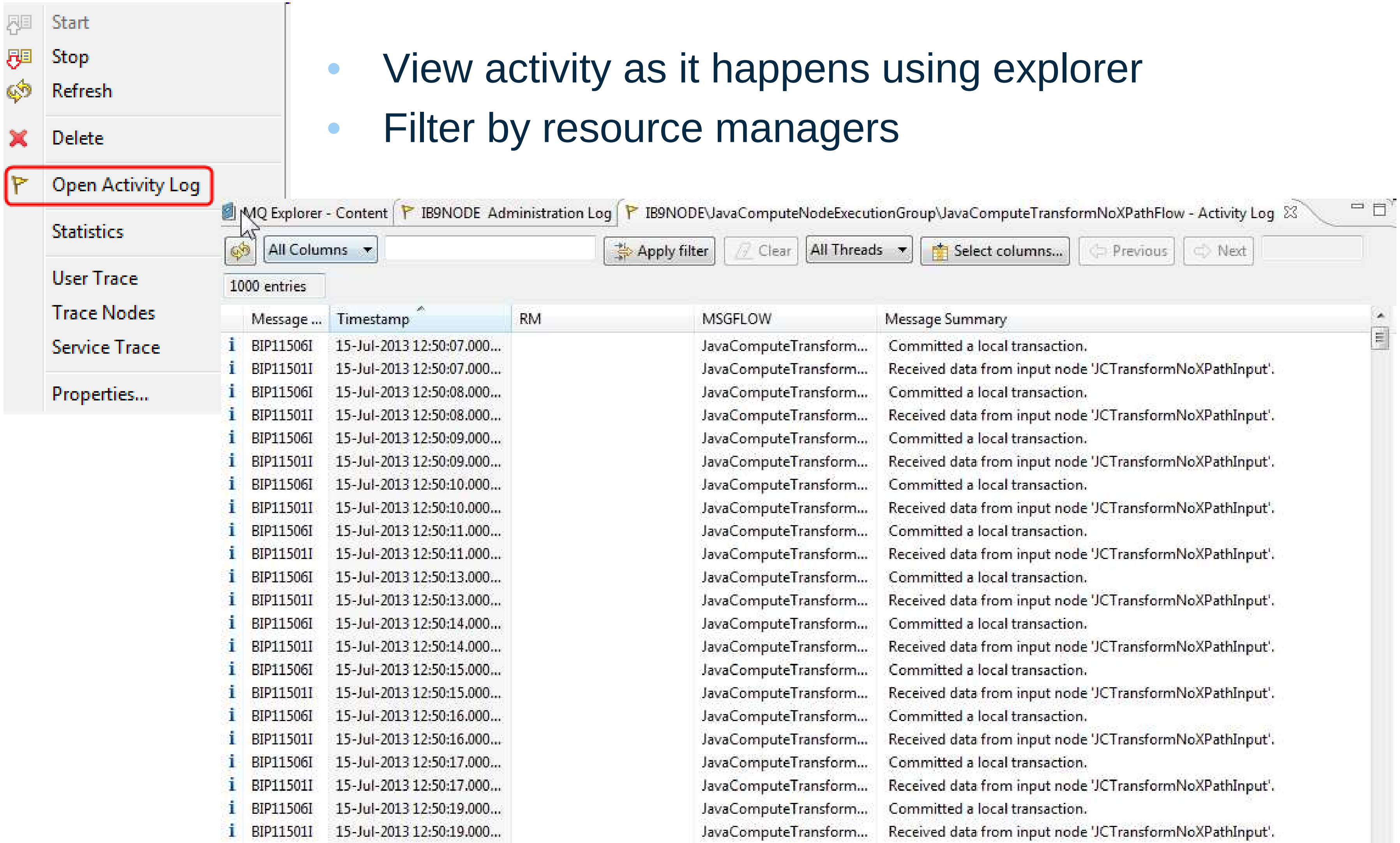


▾ Throughput per message flow for last 15 seconds. Last updated at 09:50:29 GMT Daylight Time.

Flow name	Message Rate (messages/s) ▼	Average Elapsed Time/ Invocation (ms)	Average CPU Time/ Invocation (ms)
Request	1.00	4.4	0.8
Reply	1.00	3.9	0.5
BackendReplyApp	1.00	1,001.7	1.1

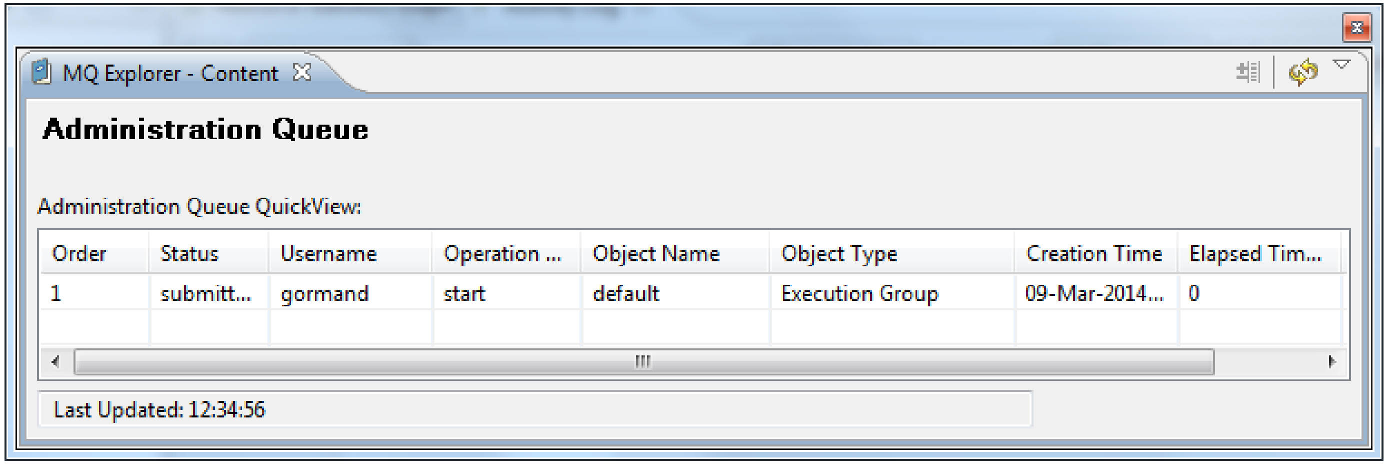
Integration Bus Explorer & Activity Log

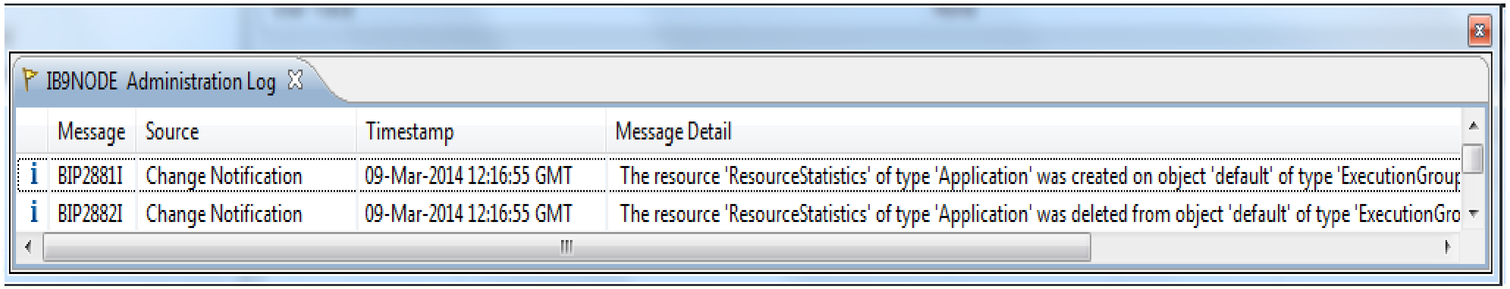
- View activity as it happens using explorer
- Filter by resource managers



Administration Queue / Log

- The tools include a lot of information that is useful to the administrator, for example:
  - **Administration queue**: What operational changes are currently pending
  - **Administration log**: What changes have been recently applied to the broker's configuration, and by whom?





Types of trace in Integration Bus

Trace is available for separate components which can be formatted to a file using:

- mqsichangetrace to enable trace
- mqsireadlog (BIPRELG) to read trace
- mqsiformatlog (BIPFMLG) to format

Types of trace available:

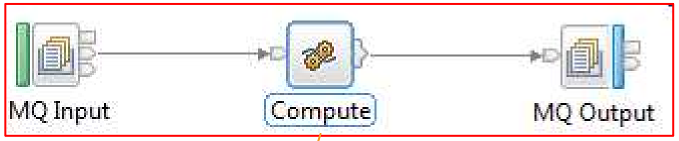
- User Trace – for you.
- Service Trace – for IBM Support
- Command Trace – for IBM Support
- CVP (Component Verification) Trace – for all

```
Aug  9 15:44 MQ91BRK.c91e9b41-3101-0000-0080-c0fdd5a003a1.trace.bin.0
Aug  9 17:28 MQ91BRK.c91e9b41-3101-0000-0080-c0fdd5a003a1.trace.bin.1
Aug  9 15:44 MQ91BRK.c91e9b41-3101-0000-0080-c0fdd5a003a1.trace.bin.2
Aug  9 15:44 MQ91BRK.c91e9b41-3101-0000-0080-c0fdd5a003a1.trace.bin.3
Aug  9 16:10 MQ91BRK.c91e9b41-3101-0000-0080-c0fdd5a003a1.userTrace.bin.0
Aug  9 17:28 MQ91BRK.c91e9b41-3101-0000-0080-c0fdd5a003a1.userTrace.bin.1
Aug  9 17:28 MQ91BRK.httplistener.trace.bin.0
Aug  9 17:28 MQ91BRK.httplistener.userTrace.bin.0
Aug  9 16:06 MQ91BRK.mqsichangeflowstats.trace.bin.0
Aug  9 16:06 MQ91BRK.mqsichangeflowstats.userTrace.bin.0
Aug  9 15:44 MQ91BRK.mqsichangetrace.trace.bin.0
Aug  9 15:44 MQ91BRK.mqsichangetrace.userTrace.bin.0
Aug  9 15:33 MQ91BRK.mqsicvp.trace.bin.0
Aug  9 17:28 MQ91BRK.mqsicvp.trace.bin.1
Aug  9 17:28 MQ91BRK.mqsicvp.userTrace.bin.0
Aug  9 15:41 MQ91BRK.mqsireadlog.trace.bin.0
Aug  9 15:41 MQ91BRK.mqsireadlog.userTrace.bin.0
Aug  9 16:11 MQ91BRK.mqsireportflowstats.trace.bin.0
Aug  9 16:11 MQ91BRK.mqsireportflowstats.userTrace.bin.0
Aug  9 17:04 MQ91BRK.mqsistop.trace.bin.0
Aug  9 17:04 MQ91BRK.mqsistop.userTrace.bin.0
Aug  9 17:28 MQ91BRK.service.trace.bin.0
Aug  9 17:28 MQ91BRK.service.userTrace.bin.0
```

User Trace Example

UserTrace tells you exactly what is happening as a message passes through an integration data flow.

See which **nodes** are being called, which **parsers** are being used, and what **ESQL** is executed.




```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    --CALL CopyEntireMessage();
    SET OutputRoot.XMLNSC.ID = InputRoot.XMLNSC.Customer.ID;
    RETURN TRUE;
END;
```

```
BIP2632I: Message received and propagated to 'out' terminal of MQ input node 'ParserStats.MQ Input'.
BIP6060I: Node 'ParserStats.MQ Input' used parser type ''Properties'' to process a portion of the incoming message of length '0' bytes beginning at offset '0'.
BIP6061I: Node 'ParserStats.MQ Input' used parser type ''MQMD'' to process a portion of the incoming message of length '364' bytes beginning at offset '0'. The parser type was selec
BIP6061I: Node 'ParserStats.MQ Input' used parser type ''XMLNSC'' to process a portion of the incoming message of length '158' bytes beginning at offset '364'. The parser type was s
BIP2537I: Node 'ParserStats.Compute': Executing statement  ''BEGIN ... END;'' at ('.ParserStats_Compute.Main', '2.2').
BIP2537I: Node 'ParserStats.Compute': Executing statement  ''SET OutputRoot.XMLNSC.ID = InputRoot.XMLNSC.Customer.ID;'' at ('.ParserStats_Compute.Main', '4.3').
BIP2539I: Node 'ImbESQLManager': Evaluating expression ''InputRoot.XMLNSC.Customer.ID'' at ('.ParserStats_Compute.Main', '4.30'). This resolved to ''InputRoot.XMLNSC.Customer.ID''.
BIP2568I: Node 'ParserStats.Compute': Copying sub-tree from ''InputRoot.XMLNSC.Customer.ID'' to ''OutputRoot.XMLNSC.ID''.
BIP2537I: Node 'ParserStats.Compute': Executing statement  ''RETURN TRUE;'' at ('.ParserStats_Compute.Main', '5.3').
BIP4015I: Message propagated to the 'out' terminal of node 'ParserStats.Compute' with the following message trees: ''.
BIP2638I: The MQ output node 'ParserStats.MQ Output' attempted to write a message to queue ''OUT'' connected to queue manager ''''. The MQCC was '0' and the MQRC was '0'.
BIP2622I: Message successfully output by output node 'ParserStats.MQ Output' to queue ''OUT'' on queue manager ''''.
```

SYSLOG / JOBLOG / STDOUT / STDERR

The Integration Bus runtime writes important operational messages to the system log:

- On z/OS, each JOBLOG includes all messages written by processes within the address space. The SYSLOG includes messages from all IIB JOBLOGs.
- On Windows, these messages are written to the event log.
- On Unix and Linux, these messages are written to the syslog.

STDOUT/STDERR may also be written to:

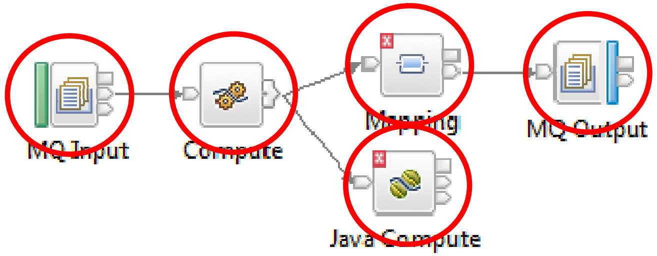
- On z/OS, each JOBLOG includes any STDOUT/STDERR written by processes within the address space.
- On Windows, the **console.txt** file in %MQSI_REGISTRY%\components\<node>\<EG UUID>
- On Unix/Linux, the **stdout** and **stderr** files in $MQSI_REGISTRY/components/<node>/<EG UUID>

External Internals

Stack based execution engine

- Useful to understand how flows execute when designing them

Start ‘MQ Input’
    Start ‘Compute’
        Start ‘Mapping’
            Start ‘MQ Output’
            Finish ‘MQ Output’
        Finish ‘Mapping’
        Start ‘Java Compute’
        Finish ‘Java Compute’
    Finish ‘Compute’
Finish ‘MQ Input’

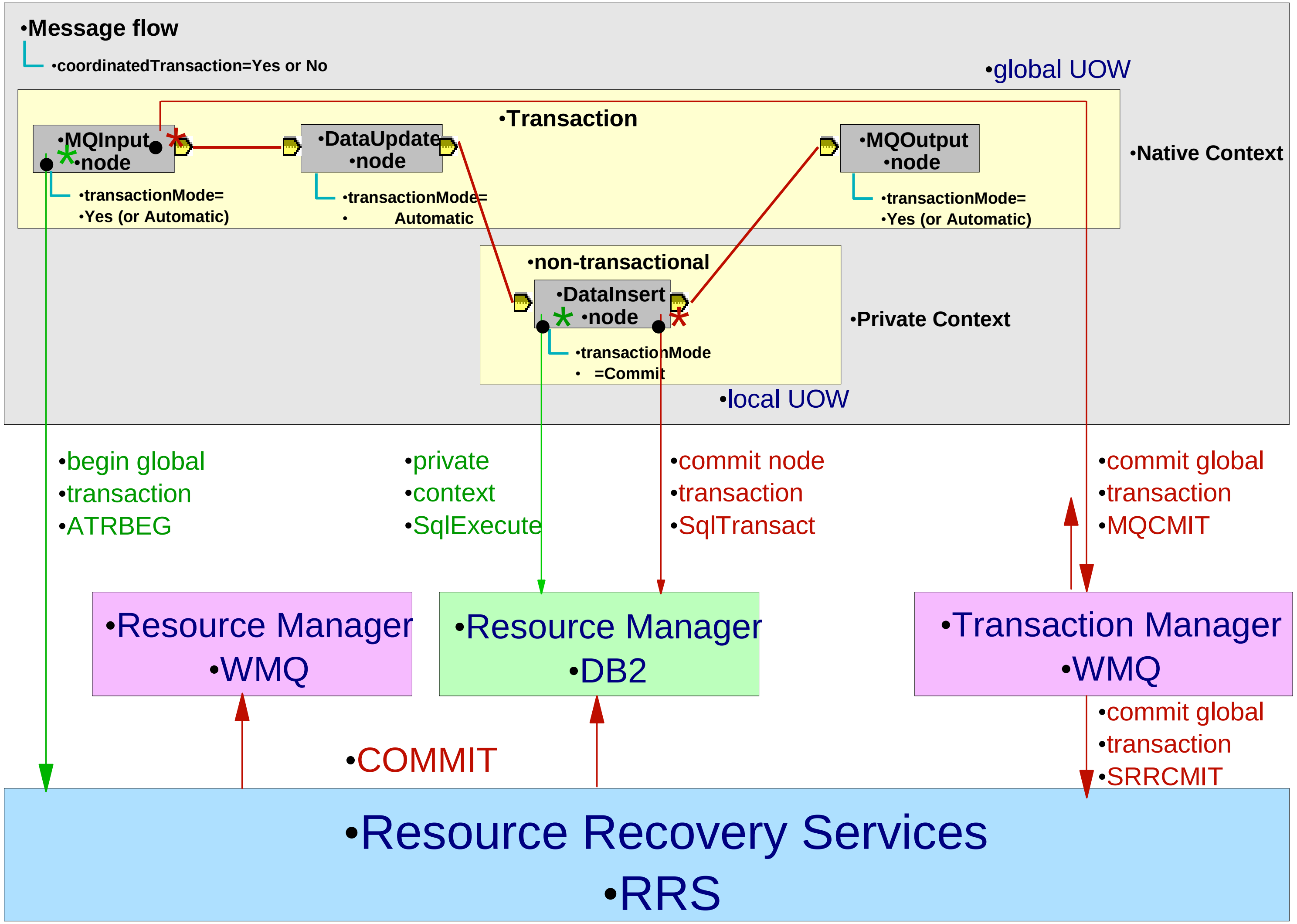


Stack:
- MQ Input
- Compute
- Mapping
- JavaCompute
- MQ Output

- Looping node connections can lead to large stack requirements
- With looping and large flows may need to increase the threadstack size
  - MQSI_THREAD_STACK_SIZE=<sizeInBytes> (Unix/Windows) (default is 1Mb)
- On z/OS the thread stack size is dynamic
  - Default size is 1Mb with 1Mb extents
  - Using the extents can impact performance if you regularly use them
  - Increase the default or extent size
    - _CEE_RUNOPTS=THREADSTACK64(ON,4M,1M)
  - Use RPTSTG(ON) option to get a report of stack sizes which were used

Transaction Model on z/OS

The z/OS broker has a global transaction model exactly as you'd expect. It is possible for nodes to elect to commit outside this transaction. RRS is used for context management & commitment control between the flows resource managers, but only when required.



Notes : Transaction Model

- **Transactional message flows are important**
  - A message flow which transforms and routes data often has a need to be transactional. That is, the message flow must complete either *all or none* of its processing. Remember, from an end-to-end application perspective, the message flow is *part* of the application.
- **Transactional data flows and data nodes.**
  - A message flow can be identified as transactional using the Coordinated Transaction checkbox on a broker assigned message flow. The intention behind this attribute is that all node operations within the message flow can be coordinated under the same, global, transaction. On z/OS, this option is always used for message flows, whether selected or not.
  - A node performs its operations within the envelope of this message flow global transaction, and can elect to be within the global transaction or not. A Transaction Mode checkbox enables this for WMQ and database nodes. Note the visibility (ACID) implications!
- **Resource Recovery Services (RRS) is *NOT* always the transaction coordinator.**
  - As message flows run in essentially a batch type address spaces, RRS is the global transaction coordinator, if required.
  - Execution groups are linked with an MQ RRS stub, so WMQ registers an interest with RRS for commitment control.
  - Specifying the keywords CONNECTTYPE=2, AUTOCOMMIT=0, MULTICONTEXT=0, and MVSATTACHTYPE=RRSAF in the initialization file BIPDSNAO enables global transaction processing.
- **RRS Context**
  - RRS Context is a concept that enables a program to have different roles. It's like one person having many ways of behaving which don't interact with each other. It means that applications can simultaneously be different things to different systems.
  - Broker flows have two contexts. A *native* context is used whenever it wants to perform the role of including node operations under the global transaction. A *private* one has the effect of excluding database node operations from a global transaction.
  - Plug-in nodes are always within the global transaction. A message flow is always in *native* context for these nodes.
- **WebSphere MQ**
  - *Transaction Mode* within a message queuing node governs whether MQPUT and MQGET operations are explicitly performed either inside or outside syncpoint on a per call basis. These nodes therefore always use the native, and never the private, RRS context.
- **Database**
  - For database nodes, *Transaction Mode* determines under which RRS context the transaction will be performed. If the node is within the global transaction, then the native context is used. For a Transaction Mode of *commit*, the private context, so that DB2 and RRS see the operation from a logically different party. These nodes commit (using SQLTransact) their operations as the node is exited.
- **Commitment Control**
  - The global transaction is begun implicitly when a resource manager communicates with RRS. The overall message transaction is committed (or backed out!) control returns to the input node. At COMMIT time, WMQ will pass control to RRS only if required.
  - RRS will call all registered resource managers (WMQ, DB2) in a two phase commit protocol to ensure a global transaction. Recall that nodes which elected for a Transaction Mode of commit had resources updated (and externally visible!) close to their point of issuing. If RRS is not required WMQ will perform the commitment control and delete any RRS interests.

Node Lifecycle (Mid-flow)

- All mid-flow nodes follow this life cycle
  - IBM written and plugin/compute nodes
- Constructor
  - Called when the node is created as the flow is initialized
- Properties are then set on the node
- onInitialize()
  - Validate configured properties
  - Called either during deployment or on broker startup.
  - If it throws an exception, deployment or startup is rolled back/stopped
    - The broker does not try to start the flow again until the broker is restarted
  - Complete tasks that will always work or always fail
    - If you need to initialize an external connection that might need to be retried, consider doing so on the first message through the flow so that the flow can retry the connection as necessary
- Evaluate
  - Called when execution of a message is required
  - Perform required node processing
  - Exceptions which are thrown and thrown back down the flow to be handled
- OnDelete
  - Called before a node is deleted
  - Use if you want the node to perform cleanup operations, for example closing sockets,
  - Should not throw exceptions

Node Lifecycle (Input)

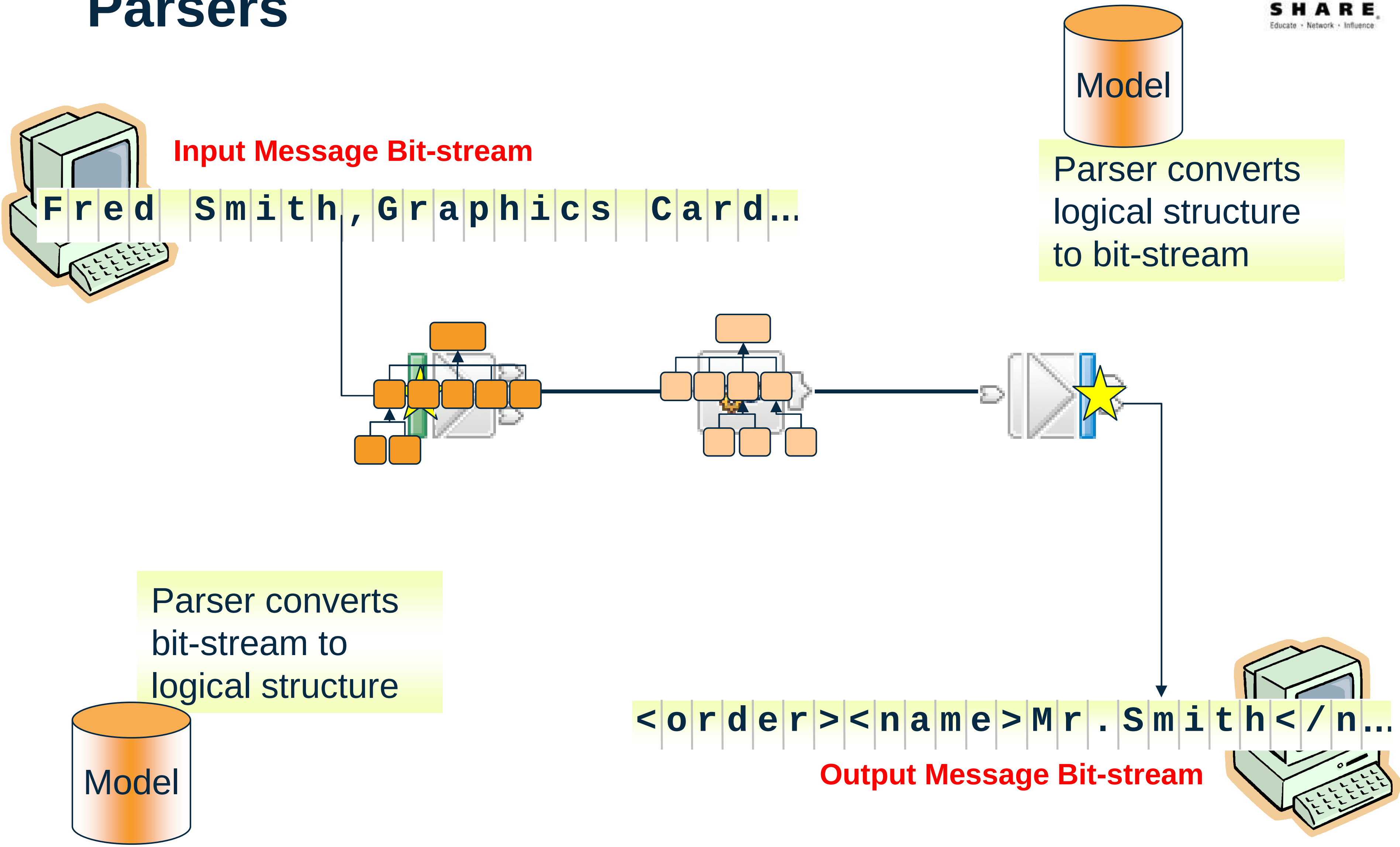
- All non-connector input nodes follow this basic lifecycle

- Constructor
  - Called when the node is created as the flow is initialized
- Properties are then set on the node
- onInitialize()

- run
  - Called by broker when we want the node to try and read data
  - Node reads data and then propagates the resulting message
  - Additional instances
    - If you want your input node to support additional instances then before propagating the message the node needs to call dispatchThread to try and dispatch another thread to read more data
  - Returns success/failure/timeout depending on result

- onDelete

Parsers


Model
Parser converts logical structure to bit-stream
Input Message Bit-stream
Fred Smith,Graphics Card...
Parser converts bit-stream to logical structure
Model
<order><name>Mr.Smith</n...
Output Message Bit-stream

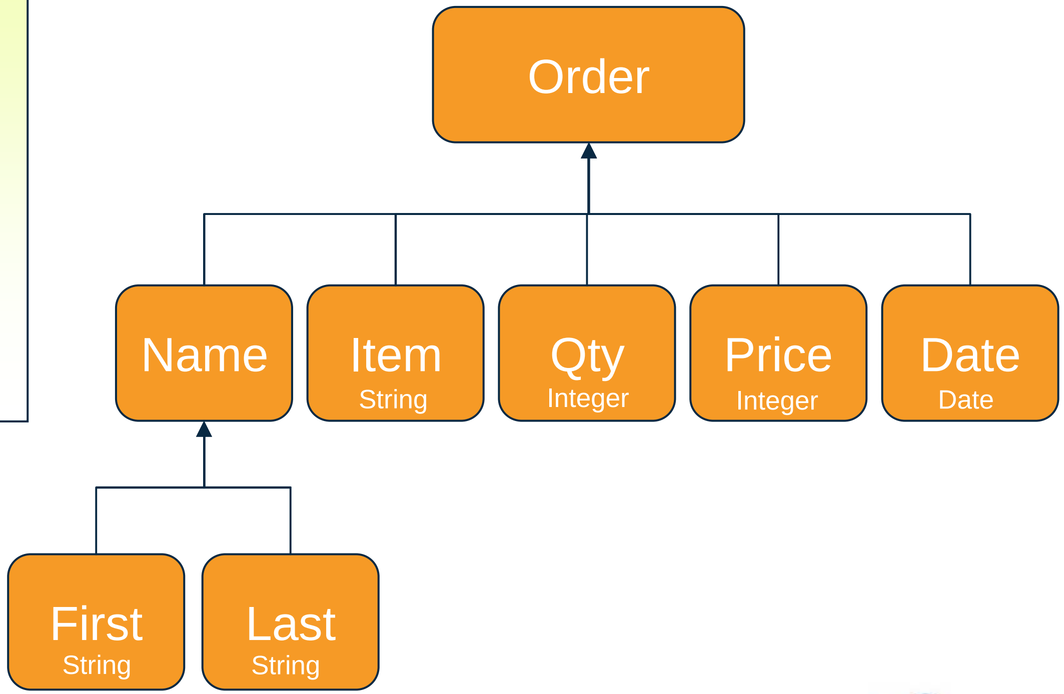
Message Modeling

Physical ⟷ Logical

```
<order>
  <name>
    <first>John</first>
    <last>Smith</last>
  </name>
  <item>Graphics Card</item>
  <quantity>32</quantity>
  <price>200</price>
  <date>07/11/09</date>
</order>
```

```
John,Smith,Graphics Card,
32,200,07/11/09
```

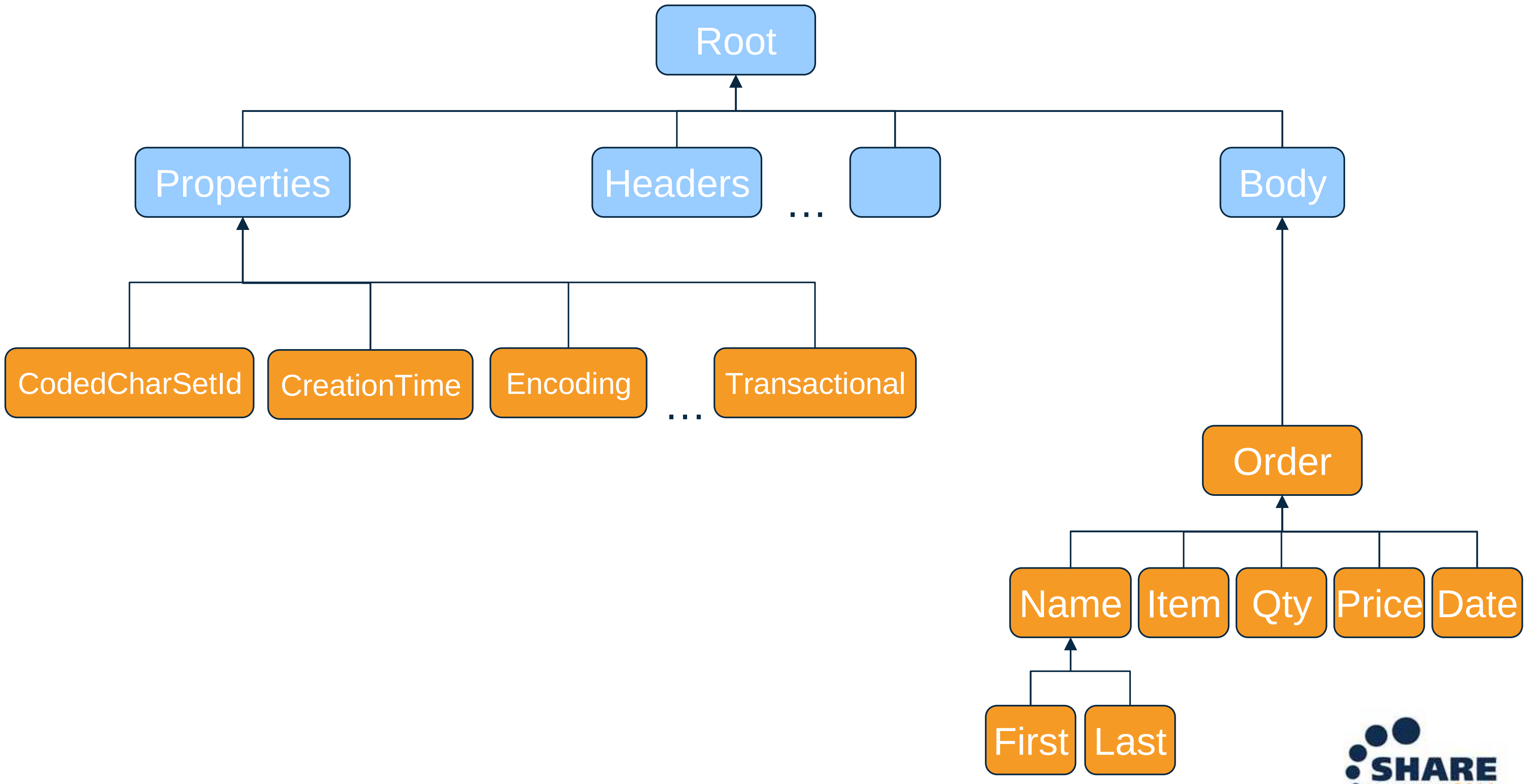
```
John Smith............
Graphics Card.........
3220020071109.........
```



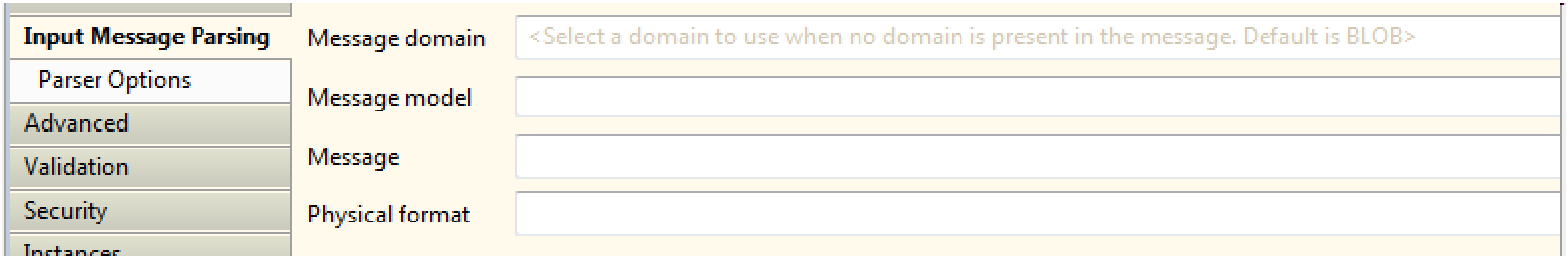
- Here is an example of how a physical data structure could be mapped to a logical tree.
  - Notice how multiple physical formats can correspond to the same logical tree. The first physical format is an XML structure that shows our Order message. The second is a comma separated value (CSV) structure of the same. The third comprises a set of fixed length fields in a custom wire format.
  - By manipulating the logical tree inside the Message Broker rather than the physical bit-stream, the nodes can be completely unaware of the physical format of the data being manipulated. It also makes it easy to introduce new message formats into the broker.
- Applications have and require diverse data formats.
  - We all know that XML is the data format that's going to solve every data processing problem that exists! We also know that "XML++", the follow-on compatible meta format that someone in a research laboratory is working on will solve all the problems we don't even know we have today! The fact is that, without wanting to appear cynical, every generation goes through this process. Surely it was the same when COBOL superseded assembler.
  - The fact is, that for historic, technical, whimsical, political, geographical, industrial and a whole host of other reasons you probably never even thought of, a hugely diverse range of data formats exist and are used successfully by a myriad of applications every second of every day. It's something that we have to live with and embrace because it isn't going to get any better any time soon.
  - The advantage WebSphere Message Broker brings by modelling all these messages is that we can rise above the message format detail; so that whether it's a tag delimited SWIFT or EDIFACT message, a custom record format closely mapping a C or COBOL data structure, or good old XML, we can talk about messages in a consistent, format independent way. Message Broker can manage this diversity.

- The Logical Message Model.
  - Reconsider messages and their structure. When we architect messages (no matter what the underlying transport technology), we concern ourselves firstly with the logical structure. For example, a funds transfer message might contain an amount in a particular currency, a transaction date and the relevant account details of the parties involved. These are the important business elements of the message; when discussing the message, we refer to these elements.
  - However, when we come to realize the message, we have to choose a specific data format. This may be driven by many factors, but we have to choose one. You may be aware of the advantages of various message formats or have your own personal favourite, or may fancy inventing a new one, but the fact remains that you have to choose a physical *wire format*. So for our transfer message, we might decide to use XML, with its elements, attributes and PCDATA (and a DTD, if we're being really exact), or we might map more closely to a C data structure modelling our message with ints, shorts, chars etc. and worry about *their* various representations(!)
  - The Logical message model provided by IBM Integration Bus allows one to describe a message in terms of a tree of elements, each of which has a (possibly user defined) type. At the message tree leaf nodes, the elements have simple types such as strings, integers, decimals, booleans etc. Moreover, elements can have various constraints and qualifiers applied to them that more fully describe them; e.g. elements might be optional, appear in a certain order or only contain certain values.

Message Tree Structure



Parser Domains



- The message domain identifies the parser that is used to parse and write instances of the message.
  - Eg: BLOB, XMLNSC, DFDL
- The remaining parts of the message template, message model, message (type), and physical format, are optional, and are used by model-driven parsers such as the DFDL parser.
- 4 Main parser types
  - Root
  - Properties
  - Header
  - Body
- Neither the Root or Properties parsers claim any of the incoming bitstream

The logical tree

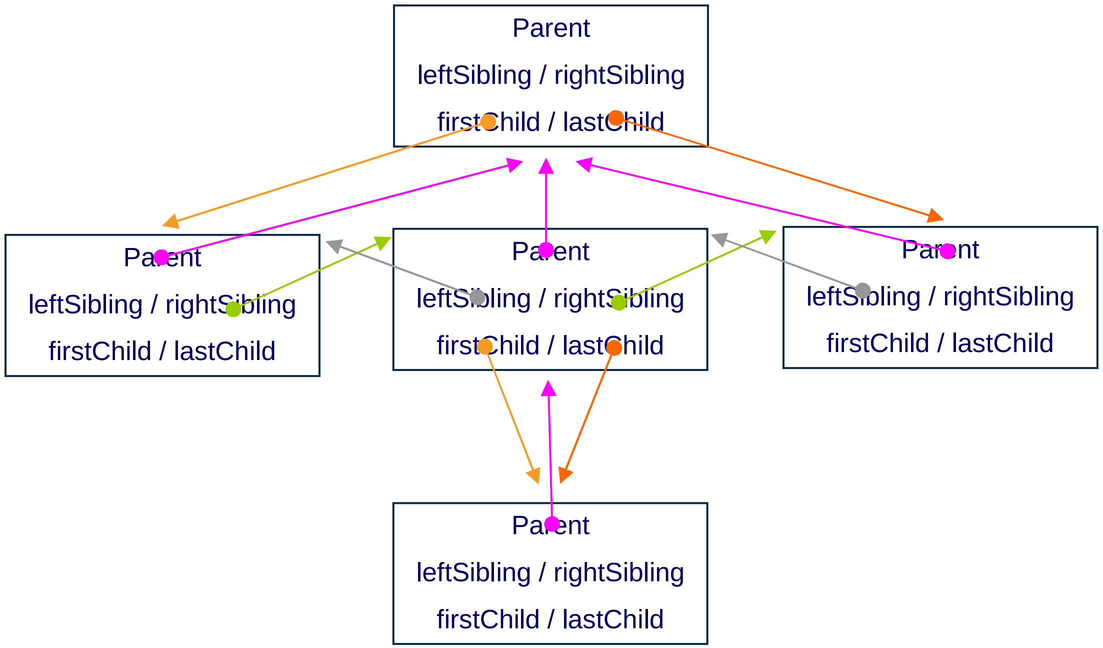
- Tree is made up of *SyntaxElement* objects which are logically linked
  - MbElement / NbElement / CciElement
- Each element contains the Name, Namespace and Value which describe that element
- Each element knows what type of element it is (Folder, Name, Value, NameValue, etc)
- Each element knows its family relationship
- Each element knows its parse state (leftComplete/rightComplete)
  - This supports partial/onDemand parsing

SyntaxElement:

Name	(String)
Namespace	(String)
Type	(Integer)
Value	(Value Type)
leftSibling	(Pointer to SyntaxElement)
firstChild	(Pointer to SyntaxElement)
parent	(Pointer to SyntaxElement)
lastChild	(Pointer to SyntaxElement)
rightSibling	(Pointer to SyntaxElement)

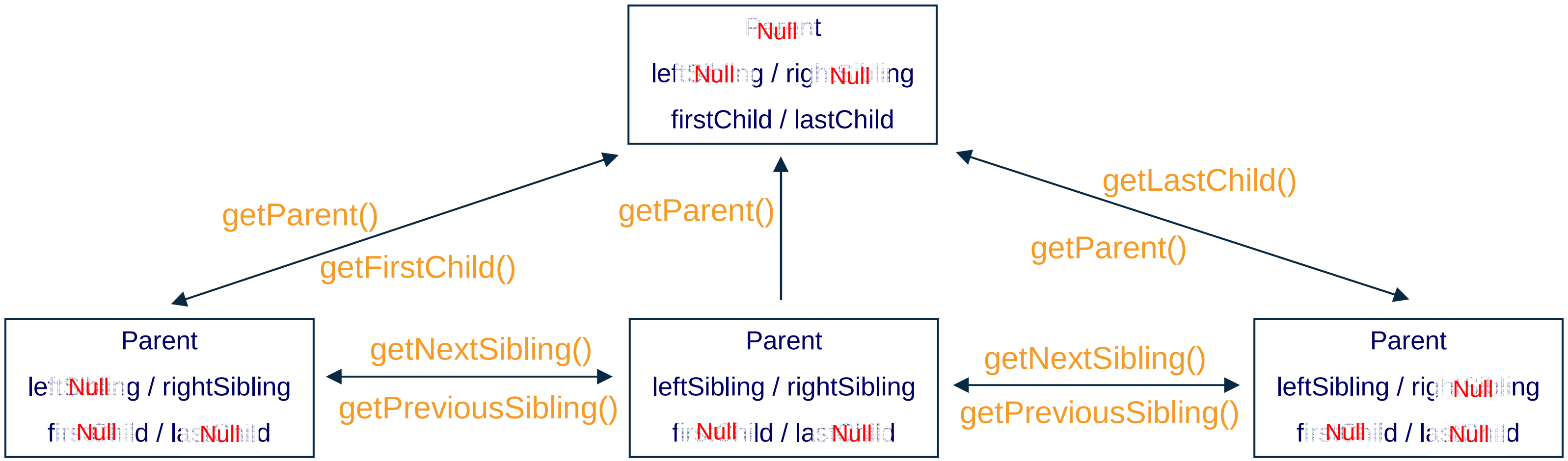
The logical tree - linkage

- The logical tree is made up of *ImbSyntaxElement* objects which are logically linked



The logical tree - navigation

- Navigation of the tree is done using a set of similar methods across all languages



Trace node output - ???

```
( ['MQROOT' : 0xed8efe0]
  (0x01000000:Name  ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
    (0x03000000:NameValue):MessageSet             = '' (CHARACTER)
      ...
    (0x03000000:NameValue):IdentityMappedIssuedBy = '' (CHARACTER)
  )
  (0x01000000:Name  ):MQMD       = ( ['MQHMD' : 0xed2bcb0]
    (0x03000000:NameValue):SourceQueue      = 'VFE.IN1' (CHARACTER)
      ...
    (0x03000000:NameValue):OriginalLength   = -1 (INTEGER)
  )
  (0x01000000:Folder):XMLNSC     = ( ['xmlnsc' : 0x1d24b30]
    (0x01000000:Folder):order = (
      ...
    )
  )
)
```

Trace node output – Root element/Parser

```
( ['MQROOT' : 0xed8efe0]
 (0x01000000:Name  ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
  (0x03000000:NameValue):MessageSet             = '' (CHARACTER)
     ...
  (0x03000000:NameValue):IdentityMappedIssuedBy = '' (CHARACTER)
 )
 (0x01000000:Name  ):MQMD       = ( ['MQHMD' : 0xed2bcb0]
  (0x03000000:NameValue):SourceQueue      = 'VFE.IN1' (CHARACTER)
    ...
  (0x03000000:NameValue):OriginalLength   = -1 (INTEGER)
 )
 (0x01000000:Folder):XMLNSC     = ( ['xmlnsc' : 0x1d24b30]
  (0x01000000:Folder):order = (
    ...
  )
 )
)
```

Trace node output - Parsers

```
( ['MQROOT' : 0xed8efe0]
 (0x01000000:Name  ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
  (0x03000000:NameValue):MessageSet            = '' (CHARACTER)
    ...
  (0x03000000:NameValue):IdentityMappedIssuedBy = '' (CHARACTER)
 )
 (0x01000000:Name  ):MQMD       = ( ['MQHMD' : 0xed2bcb0]
  (0x03000000:NameValue):SourceQueue      = 'VFE.IN1' (CHARACTER)
   ...
  (0x03000000:NameValue):OriginalLength   = -1 (INTEGER)
 )
 (0x01000000:Folder):XMLNSC     = ( ['xmlnsc' : 0x1d24b30]
  (0x01000000:Folder):order = (
   ...
  )
 )
)
```

Trace node output – Element Type

```
( ['MQROOT' : 0xed8efe0]
 (0x01000000:Name  ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
  (0x03000000:NameValue):MessageSet             = '' (CHARACTER)
     ...
  (0x03000000:NameValue):IdentityMappedIssuedBy = '' (CHARACTER)
 )
 (0x01000000:Name  ):MQMD       = ( ['MQHMD' : 0xed2bcb0]
  (0x03000000:NameValue):SourceQueue      = 'VFE.IN1' (CHARACTER)
    ...
  (0x03000000:NameValue):OriginalLength   = -1 (INTEGER)
 )
 (0x01000000:Folder):XMLNSC      = ( ['xmlnsc' : 0x1d24b30]
  (0x01000000:Folder):order = (
    ...
  )
 )
)
```

Trace node output – Element Name

```
( ['MQROOT' : 0xed8efe0]
 (0x01000000:Name  ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
  (0x03000000:NameValue):MessageSet             = '' (CHARACTER)
    ...
  (0x03000000:NameValue):IdentityMappedIssuedBy = '' (CHARACTER)
 )
 (0x01000000:Name  ):MQMD       = ( ['MQHMD' : 0xed2bcb0]
  (0x03000000:NameValue):SourceQueue      = 'VFE.IN1' (CHARACTER)
    ...
  (0x03000000:NameValue):OriginalLength   = -1 (INTEGER)
 )
 (0x01000000:Folder):XMLNSC     = ( ['xmlnsc' : 0x1d24b30]
  (0x01000000:Folder):order = (
    ...
  )
 )
)
```

Trace node output – Element Value

```
( ['MQROOT' : 0xed8efe0]
 (0x01000000:Name  ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
  (0x03000000:NameValue):MessageSet             = '' (CHARACTER)
     ...
  (0x03000000:NameValue):IdentityMappedIssuedBy = '' (CHARACTER)
 )
 (0x01000000:Name  ):MQMD       = ( ['MQHMD' : 0xed2bcb0]
  (0x03000000:NameValue):SourceQueue      = 'VFE.IN1' (CHARACTER)
    ...
  (0x03000000:NameValue):OriginalLength   = -1 (INTEGER)
 )
 (0x01000000:Folder):XMLNSC     = ( ['xmlnsc' : 0x1d24b30]
  (0x01000000:Folder):order = (
    ...
  )
 )
)
```

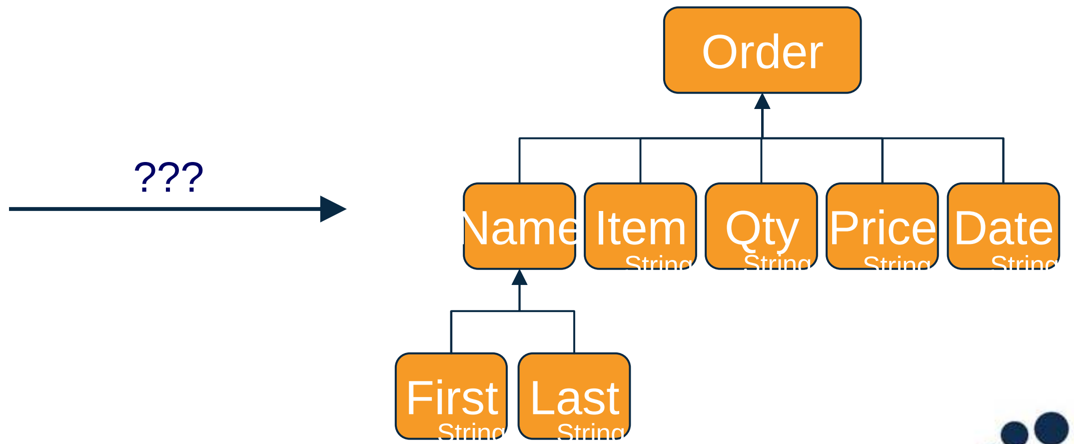
Trace node output – Element Value Type

```
( ['MQROOT' : 0xed8efe0]
  (0x01000000:Name  ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
    (0x03000000:NameValue):MessageSet              = '' (CHARACTER)
      ...
    (0x03000000:NameValue):IdentityMappedIssuedBy = '' (CHARACTER)
  )
  (0x01000000:Name  ):MQMD       = ( ['MQHMD' : 0xed2bcb0]
    (0x03000000:NameValue):SourceQueue      = 'VFE.IN1' (CHARACTER)
     ...
    (0x03000000:NameValue):OriginalLength   = -1 (INTEGER)
  )
  (0x01000000:Folder):XMLNSC     = ( ['xmlnsc' : 0x1d24b30]
    (0x01000000:Folder):order = (
     ...
    )
  )
)
```

Building the logical tree – aka parsing!

- For XML it is easy to visually see how we get from the input message to the logical model
- But how?
- The parser fires events back to a handler and the handler then creates the tree
  - Java SAX like parsing
  - DFDL parser follows same model
  - This model allows of OnDemand/partial parsing
    - Reduced memory as you do not always require all of the logical tree in memory

```
<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
```



Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)

<Order>

<Name>

<First>John</First>

<Last>Smith</Last>

</Name>

<Item>Graphics Card</Item>

<Qty>32</Qty>

<Price>200</Price>

<Date>07/11/09</Date>

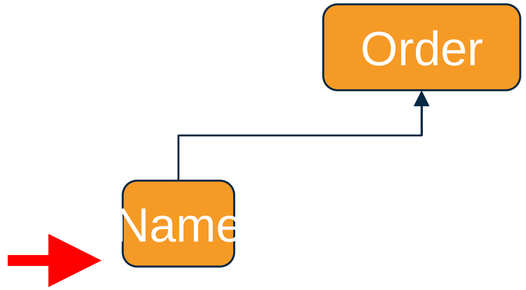
</Order>

Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)
    - startElement (Name)

<Order>
<Name>
<First>John</First>
<Last>Smith</Last>
</Name>
<Item>Graphics Card</Item>
<Qty>32</Qty>
<Price>200</Price>
<Date>07/11/09</Date>
</Order>



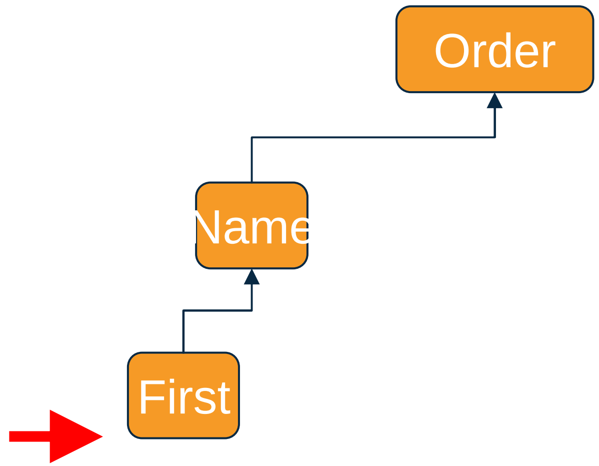
SHARE in Pittsburgh 2014

Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)
    - startElement (Name)
      - startElement (First)

```
<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
```

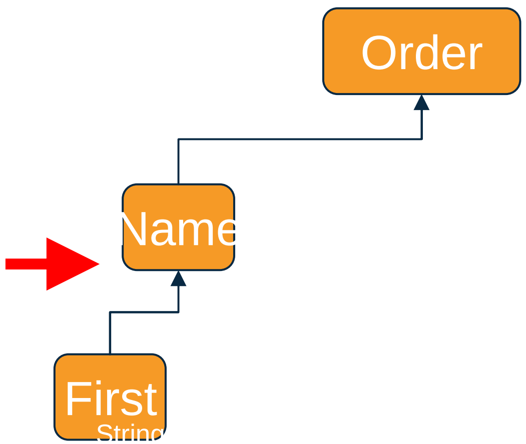


Building the logical tree – aka parsing!

Events:

- `startDocument`
  - `startElement (Order)`
    - `startElement (Name)`
      - `startElement (First)`
        - `elementValue(John)`
      - `endElement`

<Order>
<Name>
<First>John</First>
<Last>Smith</Last>
</Name>
<Item>Graphics Card</Item>
<Qty>32</Qty>
<Price>200</Price>
<Date>07/11/09</Date>
</Order>

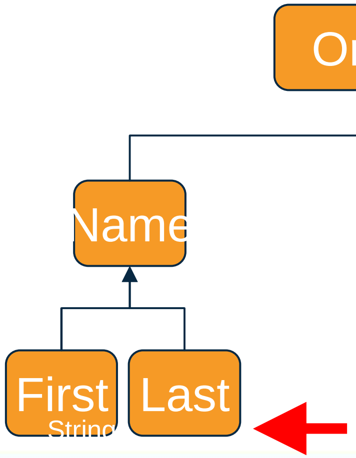


Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)
    - startElement (Name)
      - startElement (First)
        - elementValue(John)
      - endElement
      - startElement (Last)

<Order>
<Name>
<First>John</First>
<Last>Smith</Last>
</Name>
<Item>Graphics Card</Item>
<Qty>32</Qty>
<Price>200</Price>
<Date>07/11/09</Date>
</Order>

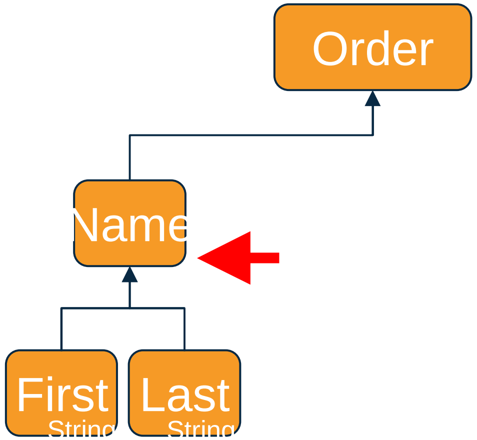


Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)
    - startElement (Name)
      - startElement (First)
        - elementValue(John) (String)
      - endElement
      - startElement (Last)
        - elementValue(Smith) (String)
      - endElement

```
<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
```

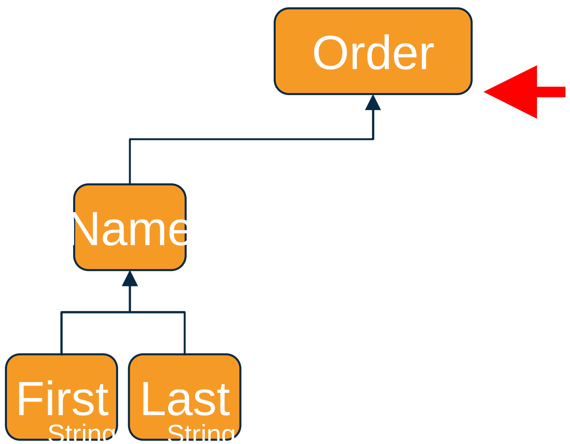


Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)
    - startElement (Name)
      - startElement (First)
        - elementValue(John) (String)
      - endElement
      - startElement (Last)
        - elementValue(Smith) (String)
      - endElement
    - endElement

```
<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
```

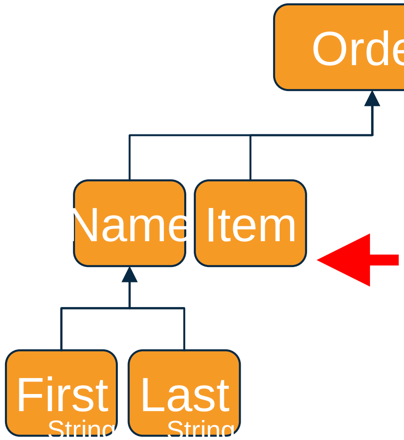


Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)
    - startElement (Name)
      - startElement (First)
        - elementValue(John) (String)
      - endElement
      - startElement (Last)
        - elementValue(Smith) (String)
      - endElement
    - endElement
    - startElement (Item)

```
<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
```

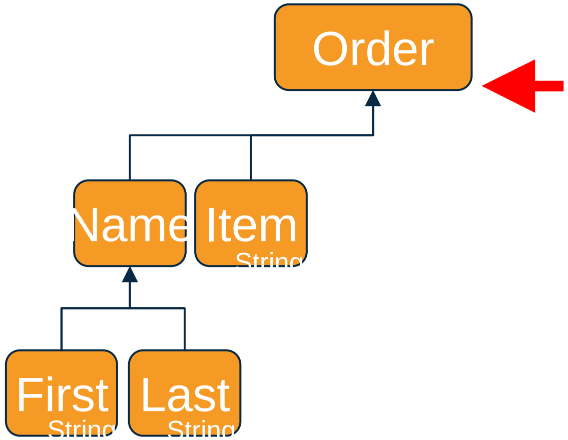


Building the logical tree – aka parsing!

Events:

- startElement (Name)
  - startElement (First)
    - elementValue(John) (String)
  - endElement
  - startElement (Last)
    - elementValue(Smith) (String)
  - endElement
- endElement
- startElement (Item)
  - elementValue(Graphics Card) (String)
- endElement

```
<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
```



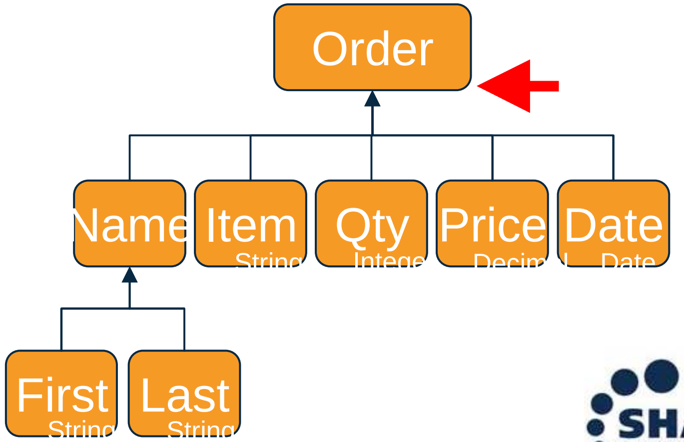
Building the logical tree – aka parsing!

Events:

... Repeat 3 more times for Qty, Price and Date ...

- startElement (Qty)
  - elementValue(32) (Integer)
- endElement
- startElement (Price)
  - elementValue(Smith) (Decimal)
- endElement
- startElement (Date)
  - elementValue (07/11/09) (Date)
- endElement

```
<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
```



Building the logical tree – aka parsing!

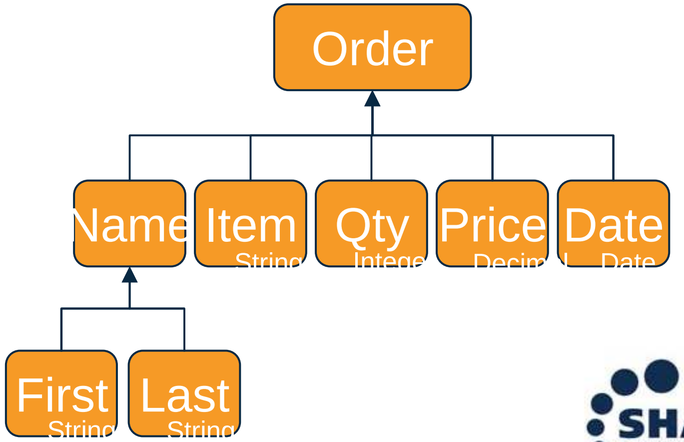
Events:

- `startElement (Date)`
  - `elementValue (07/11/09) (Date)`
- `endElement`

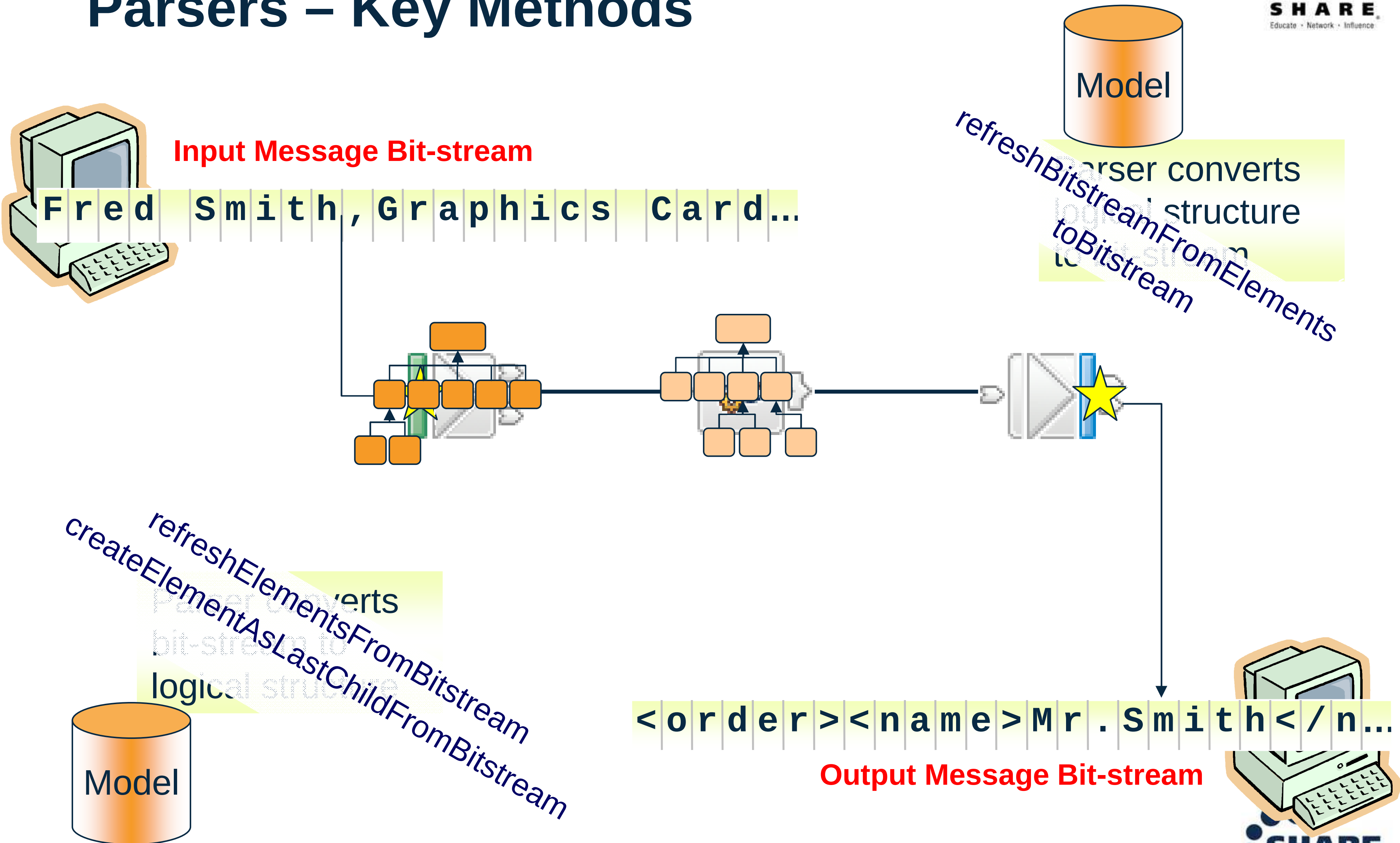
  – `endElement`

- `endDocument`

```
<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
```



Parsers – Key Methods



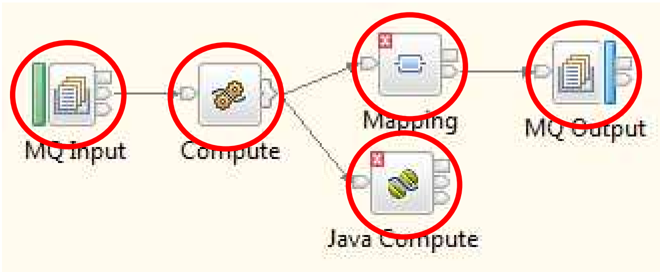
Parsers – Key Methods

- Applies to all parsers, IBM written and plugin parsers
- refreshElementsFromBitstream
  - Turns the bitstream into the logical tree
  - Takes various options
    - Encoding and ccsid to use for the parse
    - Message Set, Type & Format to use for the parse
    - Parser options to apply to the parse, such as validation
    - The parser remembers the options it was initiated with
  - Can be driven from createElementAsLastChildFromBitstream calls in transformation

- refreshBitstreamFromElements
  - Turns the logical tree into a bitstream
  - Takes various options that match those on the refreshElementsFromBitstream call
  - Can be driven from toBitstream calls in transformation

- There is an optimization present in the parsers so that if a parser is called with exactly the same options on the refreshBitstreamFromElements call as it initialized with on the refreshElementsFromBitstream call, and the logical tree hasn't been modified then the input bitstream is returned
  - This helps support simple pass-through and routing scenarios

Parser Internals – reuse example



1. MQInput Node

2. MQInput Node
   Compute Node

3. MQInput Node
   Compute Node
   Mapping Node

4. MQInput Node
   Compute Node
   Mapping Node
   MQOutput Node

5. MQInput Node
   Compute Node
   Mapping Node

6. MQInput Node
   Compute Node

7. MQInput Node
   Compute Node

8. MQInput Node
   Compute Node
   JavaCompute Node

9. MQInput Node
   Compute Node

10. MQInput Node

Flow pool
Node pool
Parser

Notes:

- 1: 7 Parsers are available in the flow pool
- 2: The Compute node creates a message using 2 parsers
- 3: The Mapping node creates a message using 2 parsers
- 4: The MQOutput node creates a message using 1 parser
- 5-6: As the stack unwinds parsers are returned for reuse
- 7: The Compute node creates a new message using 1 of the previously used parsers
- 8: Java Compute node creates a new message using 2 previously used parsers
- 9-10: As the stack unwinds parsers are returned for reuse

Parser Internals - reuse

- Each Message Flow Instance has its own pool of parsers
- Nodes in a flow ‘borrow’ parsers from the flow pool
- Message’s created in nodes use the node ‘borrowed’ parsers
- If a Message requires a new parser it asks the Node for one who in turn ask the flow pool to borrow one
- If the flow pool does not have a free parser it creates a new one
- When a node goes off of the stack the parsers are returned to the flow pool for reuse.
- Only in this instance are parsers returned to the pool for reuse mid flow
  – example next slide
- At the end of processing a message the flow pool is reset meaning all parsers are reset and NOT DELETED
- This means their memory is still in use
- The parsers are only deleted when the flow is stopped or undeployed

Parser Internals - MQSI_FREE_MASTER_PARSERS

- Each Message Flow Instance has its own pool of parsers
- Nodes in a flow ‘borrow’ parsers from the flow pool
- Message’s created in nodes use the node ‘borrowed’ parsers
- If a Message requires a new parser it asks the Node for one who in turn ask the flow pool to borrow one
- If the flow pool does not have a free parser it creates a new one
- When a node goes off of the stack the parsers are returned to the flow pool for reuse.
- Only in this instance are parsers returned to the pool for reuse mid flow
  – example next slide
- At the end of processing a message the Master ImbMessageGroup is reset and ALL PARSERS are DELETED and not reset
- This means their memory is returned to the OS

Parser Internals - Element Pools

- Each Parser has a pool of elements
- As a parser is reused the elements in the pool are reused
- The elements owned by a parser are kept until the parser is deleted
- When a parser is reused it may not be used for the same purpose as last time
  - ie, one use maybe on the input message and another time for the output message
- Each reuse may require a different number of elements
- Over time the element pool for each parser of the same domain will grow until it is the maximum size required
  - This is the plateau'ing effect we sometimes describe in PMRs

Parser resource statistics

- Use the Parsers statistics to see how much resource is being used by the message trees and bit streams that these parsers own.

Measurements	Description
Threads	The number of message flow threads that contributed to the statistics for a message flows parser type accumulation.
ApproxMemKB	The approximate amount of user data-related memory used for the named message flow parser type. It is not possible to calculate the exact amount of memory used by a parser.
MaxReadKB	Shows the largest bit stream parsed by the parser type for the named message flow.
MaxWrittenKB	Shows the largest bit stream written by the parser type for the named message flow.
Fields	Shows the number of message fields associated with the named message flow parser type. These fields are retained by the parser and are used for constructing the message trees.
Reads	The number of successful parses that were completed by the named message flow parser type.
FailedReads	The number of failed parses that occurred for the named message flow parser type.
Writes	The number of successful writes that were completed by the named message flow parser type.
FailedWrites	The number of failed writes that occurred for the named message flow parser type.

Parser Resource Statistics - Interpretation

MB8BROKER Administration Log | default Resources Statistics (Snapshot time 18:23:08 - 18:23:28) | Progress

DotNet GC | CORBA | ConnectDirect | FTEAgent | FTP | File | GlobalCache | JDBCConnectionPools | JMS | JVM | ODBC | Parsers | SOAPInput | Security | Sockets | TCPIPClien

name	Threads	ApproxMem...	MaxReadKB	MaxWrittenKB	Fields	Reads	FailedReads	Writes	FailedWrites
summary	1	119.77	0.53	0.53	159	6	0	4	0
VFE_1_Flow.MQMD	1	15.97	0.36	0.43	62	2	0	1	0
VFE_1_Flow.MQROOT	1	55.89	0.53	0.00	7	1	0	1	0
VFE_1_Flow.Properties	1	31.94	0.53	0.00	72	1	0	1	0
VFE_1_Flow.XMLNSC	1	15.97	0.18	0.53	18	2	0	1	0
[Deleted]	0	0.00	0.00	0.00	0	0	0	0	0
[Administration]	3	223.56	3.97	0.00	201	24	0	0	0

```
(0x01000000:Folder):XM 1 C     = ( ['xmlnsc' : 0x1cd2360]
  (0x01000000:F 2 ):order = (
    (0x01000000:Fold 3 :name     = (
      (0x03000000:PCDataFie 4 t = 'John' (CHARACTER)
      (0x03000000:PCDa 5 ):last  = 'Smith' (CHARACTER)
    )
    (0x03000000:PCDataF 6 em     = 'Graphics Card'
    (CHARACTER)
    (0x03000000:PCDat 7 :quantity = '32' (CHARACTER)
    (0x03000000:PCDataField 8 e     = '200' (CHARACTER)
    (0x03000000:PCDataF 9 date     = '07/11/09'
    (CHARACTER)
  )
)
```

```
(0x01000000:Fo 10 XMLNSC     = ( ['xmlnsc' : 0x342ba2f8]
  (0x0100 11 Folder):order = (
    (0x01000000 12 ):name     = (
      (0x03000000:PC 13 ield):first = 'John' (CHARACTER)
      (0x03000000:PCD 14 ld):last  = 'Smith' (CHARACTER)
    )
    (0x03000000:PCDa 15 ):item     = 'Graphics Card'
    (CHARACTER)
    (0x03000000:P 16 Field):quantity = '32' (CHARACTER)
    (0x03000000:PCDat 17 ):price     = '200' (CHARACTER)
    (0x03000000:PCD 18 ld):date     = '07/11/09'
    (CHARACTER)
  )
)
```

Parser Resource Statistics - Interpretation

- Using our earlier example flow
- ESQL changed from this:

```
SET OutputRoot = InputRoot;
```

- To:

```
CALL CopyMessageHeaders();
SET OutputRoot.XMLNSC.order = InputRoot.XMLNSC.order;
```

- Previously a parser-to-parser copy took place and because no modifications have taken place a new set of parsers were initialised with the original bitstream hence a read of 2 as a new parse takes place
- With the new code a tree copy is taking place having navigated both sides to the order element – now only 1 read taking place
- Parser stats can be very powerful when analysing changes to message flows with high parser memory usage

MB8BROKER Administration Log | default Resources Statistics (Snapshot time 00:48:28 - 00:48:48) | Progress

ConnectDirect | FTEAgent | FTP | File | GlobalCache | JDBCConnectionPools | JMS | JVM | ODBC | Parsers | SOAPInput | Security | Sockets | TCPIPClientNodes | TCPIPServerNodes

name	Threads	ApproxMem...	MaxReadKB	MaxWrittenKB	Fields	Reads	FailedReads	Writes	FailedWrites
summary	1	103.80	0.54	0.51	158	5	0	4	0
VFE_1_Flow.MQMD	1	15.97	0.36	0.43	62	2	0	1	0
VFE_1_Flow.MQROOT	1	55.89	0.54	0.00	7	1	0	1	0
VFE_1_Flow.Properties	1	15.97	0.54	0.00	70	1	0	1	0
VFE_1_Flow.XMLNSC	1	15.97	0.18	0.51	19	1	0	1	0
[Deleted]	3	0.00	0.54	0.53	525	20	1	12	0
[Administration]	3	223.56	7.31	0.00	217	28	0	0	0

Partial parsing

- Partial parsing is where only part of the message is parsed at a time, and only if required
- Utilised correctly this can reduce memory usage and increase performance
- Default parse mode is “on Demand” which means only parse as far as you need to, to satisfy the current request.
- In our example message if we only needed to read the ‘name’ element to route the message and didn’t need to make a modification then as long as we only referenced as far as the name element then we wouldn’t need to parse all of the message

- With our example flow
  - If we disable the trace nodes then we will see the field count in the parser stats reduce as we haven’t needed to parse all of the message
  - Thus partial parsing has been utilised

```
<order>
  <name>
    <first>John</first>
    <last>Smith</last>
  </name>
  <item>Graphics Card</item>
  <quantity>32</quantity>
  <price>200</price>
  <date>07/11/09</date>
</order>
```

Parser Resource Statistics - Interpretation

- Use Parser statistics to understand memory costs associated with processing messages.
  - This is a simple XML file, parsed and serialised using **XMLNSC**.
  - The actual size of the file is **118** bytes (matches **0.12KB** reported).
  - Not all fields have been parsed, as the flow has parse on demand.

```
<Customer>
  <FirstName>Joe</FirstName>
  <LastName>Bloggs</LastName>
  <ID>1234567890123456789</ID>
</Customer>
```

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    CALL CopyEntireMessage();
    RETURN TRUE;
END;
```

DotNet App Domains | CICS | DotNet GC | CORBA | ConnectDirect | DecisionServices | FTEAgent | FTP | File | GlobalCache | JDBCConnectionPools

ExecutionGroup	name	Threads	ApproxMemKB	MaxReadKB	MaxWrittenKB	Fields	Reads	FailedReads	Writes	FailedWrites
default	summary	1	111.78	0.47	0.47	17	7	0	4	0
default	ParserStats.MQMD	1	15.97	0.36	0.43	2	2	0	1	0
default	ParserStats.MQROOT	1	55.89	0.47	0.00	7	1	0	1	0
default	ParserStats.Properties	1	23.95	0.00	0.00	6	2	0	1	0
default	ParserStats.XMLNSC	1	15.97	0.12	0.47	2	2	0	1	0
default	[Deleted]	0	0.00	0.00	0.00	0	0	0	0	0
default	[Administration]	2	63.88	0.52	0.00	73	4	0	0	0

More Parser Resource Statistics - Interpretation

- The same input message as before, but different ESQL. This time the ESQL statement refers to the last element (Customer.ID).
- The parser statistics show how many more fields are read.

1 (XMLNSC)

```
<Cus 2 >
  <First 3 Joe</FirstName>
  <L 4 me>Bloggs</LastName>
  <ID>123 5 90123456789</ID>
</Customer>
```

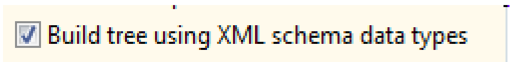
```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    --CALL CopyEnti 6 7 ;
    SET OutputRoot.XMLNSC.ID = InputRoot.XMLNSC.Customer.ID;
    RETURN TRUE;
END;
```

DotNet App Domains | CICS | DotNet GC | CORBA | ConnectDirect | DecisionServices | FTEAgent | FTP | File | GlobalCache | JDBCConnectionPools | JMS | JVM | ODBC | Parsers

name	Threads	ApproxMemKB	MaxReadKB	MaxWrittenKB	Fields	Reads	FailedReads	Writes	FailedWrites
summary	1	95.81	0.47	0.03	24	4	0	3	0
ParserStats.MQMD	1	7.98	0.36	0.00	1	1	0	0	0
ParserStats.MQROOT	1	55.89	0.47	0.00	7	1	0	1	0
ParserStats.Properties	1	15.97	0.00	0.00	9	1	0	1	0
ParserStats.XMLNSC	1	15.97	0.12	0.03	7	1	0	1	0
[Deleted]	0	0.00	0.00	0.00	0	0	0	0	0
[Administration]	2	71.86	4.88	0.00	144	8	0	0	0

and finally … Typed trees

- When parsing is performed using a schema or message set the element types can be set to be appropriate types as defined in the model
  - MRM / DFDL - automatically typed based on the model
  - XMLNSC – requires content + validation enabled and “Build tree using XMLNSC schema data types” selected



Before:

```
(0x01000000:Folder):XMLNSC     = ( ['xmlnsc' : 0x1cd2360]
  (0x01000000:Folder):order = (
    (0x01000000:Folder     ):name     = (
      (0x03000000:PCDataField):first = 'John' (CHARACTER)
      (0x03000000:PCDataField):last  = 'Smith' (CHARACTER)
    )
    (0x03000000:PCDataField):item     = 'Graphics Card'
   (CHARACTER)
    (0x03000000:PCDataField):quantity = '32' (CHARACTER)
    (0x03000000:PCDataField):price    = '200' (CHARACTER)
    (0x03000000:PCDataField):date     = '07/11/09'
   (CHARACTER)
  )
)
```

After:

```
(0x01000000:Folder):XMLNSC     = ( ['xmlnsc' : 0x3441b838]
  (0x01000000:Folder):order = (
    (0x01000000:Folder     ):name     = (
      (0x03000000:PCDataField):first = 'John' (CHARACTER)
      (0x03000000:PCDataField):last  = 'Smith' (CHARACTER)
    )
    (0x03000000:PCDataField):item     = 'Graphics Card'
   (CHARACTER)
    (0x03000000:PCDataField):quantity = 32 (DECIMAL)
    (0x03000000:PCDataField):price    = 200 (DECIMAL)
    (0x03000000:PCDataField):date     = DATE '2007-11-09'
   (DATE)
  )
)
```

This was session 16200 - The rest of the week ……

	Monday	Tuesday	Wednesday	Thursday	Friday
08:30			Application programming with MQ verbs	The Dark Side of Monitoring MQ - SMF 115 and 116 Record Reading and Interpretation	CICS and MQ - Workloads Unbalanced!
10:00					
11:15	Introduction to MQ	What's New in IBM Integration Bus & WebSphere Message Broker	MQ – Take Your Pick Lab	Using IBM WebSphere Application Server and IBM WebSphere MQ Together	
12:15					
01:30		All about the new MQ v8	MQ Security: New v8 features deep dive	New MQ Chinit monitoring via SMF	
03:00	MQ Beyond the Basics	MQ & DB2 – MQ Verbs in DB2 & InfoSphere Data Replication (Q Replication) Performance	What's wrong with MQ?	IIIB - Internals of IBM Integration Bus	
04:15	First Steps with IBM Integration Bus: Application Integration for a new world	MQ for z/OS v8 new features deep dive	MQ Clustering - The Basics, Advances and What's New in v8		

Questions?