



## The Enterprise Service Bus


*WebSphere ESB*

*WebSphere Message Broker*

*WebSphere Service Registry and Repository*

Don Bagwell  
IBM Washington Systems Center  
dbagwell@us.ibm.com

© 2007 IBM Corporation



**This slide intentionally left blank**

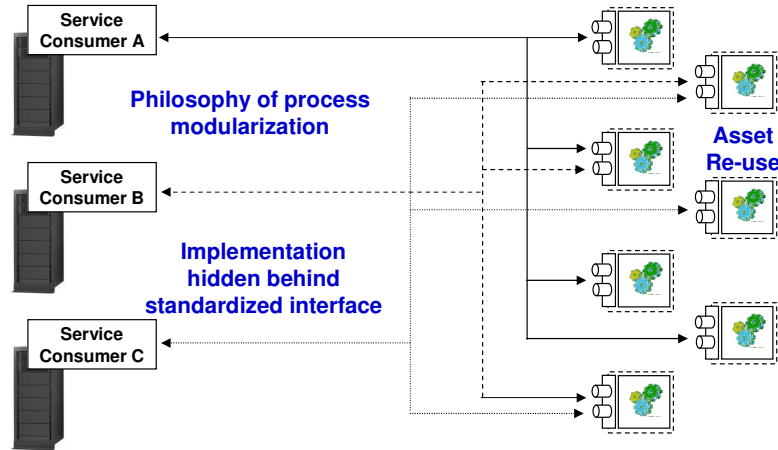
**2**

**IBM Americas Advanced Technical Support**  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

## The Value of Loosely-Coupled Services

It allows the flexible re-use of assets. Construct business processes by using these services in the order needed:



This is good -- it achieves at least a few key things:

- Establishment of a mindset towards reusable, service-oriented design
- The creation of an inventory of reusable service assets
- The hiding of complex implementation details behind a standardized interface

But this *by itself* does not solve the complexity issue ...

3

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

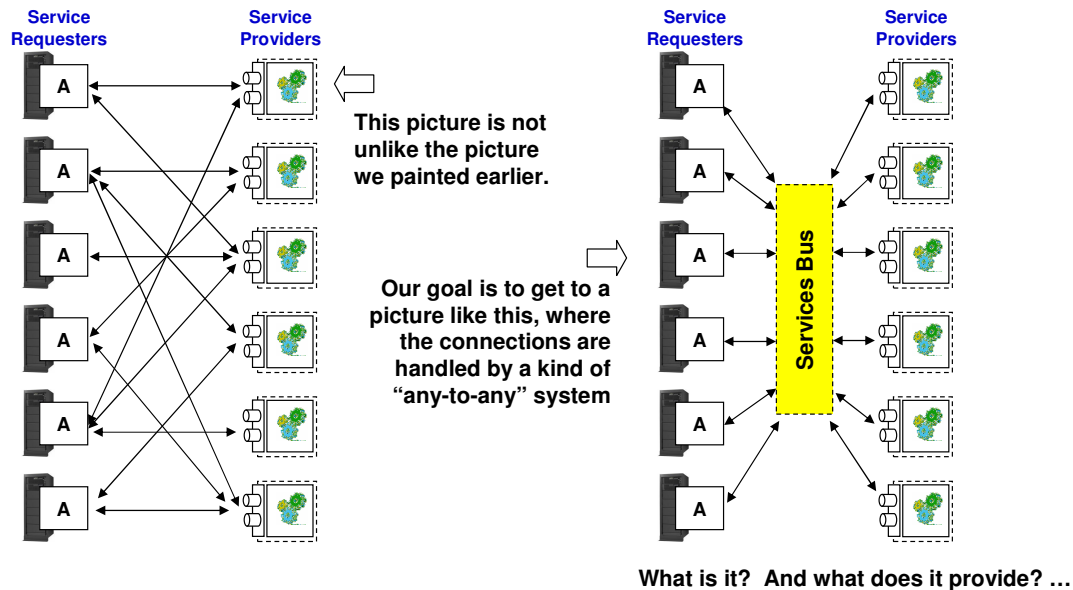
© 2007 IBM Corporation

We've spent a good deal of time focusing on the role of Web Services. We've seen that creating a library of reusable services can lead to greater flexibility. Having users of those services "loosely coupled" (that is, integration with them not buried deep in the source code, or in some inflexible properties file) can assist in this flexibility. By hiding the complexity of the implementation behind a standardized interface, we can shield users from things they don't need to worry about and expose only those things really applicable to the service.

This is all good. It is a good step towards SOA. But by itself it is not SOA. Because it does not itself solve the basic complexity issue.

## Still Point-to-Point

Services are only the first step. If only a few, then easy to manage. But when the number increases, the complexity increases as well. We need to address the point-to-point nature.



4

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

Web Services, as we've described them, is still a "point-to-point" architecture. And without any intermediary intelligence, the interconnections between requesters and providers can become nearly as complex as the picture we painted to start this discussion. Things are a little better in that the interconnections are not tightly-coupled. There's some flexibility. But the picture can still become challenging as the number of users and services increases.

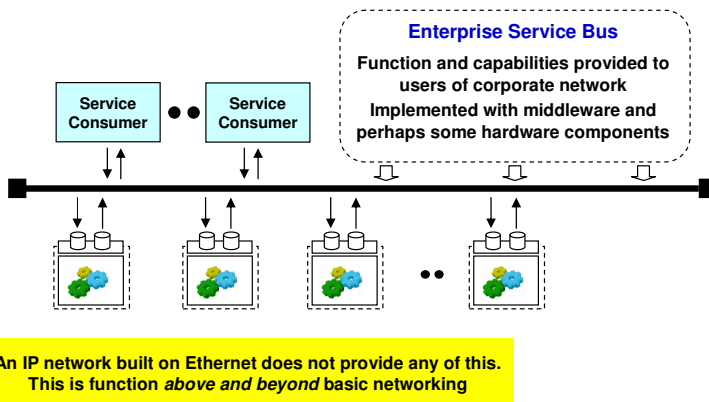
Our goal is to get to something like what we're showing on the right. Ideally all users would connect to "something" in the middle, and that "something" would connect requester to provider using intelligence. The picture then gets a lot less complex.

We have that "something" labeled as a "Services Bus" in the picture. But what exactly is that?

## The ESB -- What It Is and What it Provides

Here's the picture from our Introduction presentation. Key point is that the ESB is function mapped on top your existing network infrastructure.

- **Messaging services**  
Support different message types; content-based routing; guarantee message delivery.
- **Management services**  
Monitor performance; enforce SLA
- **Interface services**  
Support web services standards and provider "adapters" for non-standard interfaces
- **Mediation services**  
Transform messages between formats.
- **Security services**  
Encryption, authentication, authorization



Even with this it's a bit of a slippery concept. Let's take a look at one more "concept picture" then introduce IBM's ESB products.

A common initial use of the ESB ...

5

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

We have tried, throughout these presentations, to offer a physical view of things whenever possible. Logical views have their place, but for some of us it's helpful to see things in a more practical light to really get a handle on what something is.

This picture is one attempt at showing a semi-physical representation of the Enterprise Service Bus -- the ESB. The key point we're trying to make is that the ESB is not a separate physical network, but rather middleware function mapped on top your existing corporate network. Your service consumers and service providers end up connecting *through* this new middleware function, but they'll do so *across* your existing corporate network infrastructure.

The notion here is that an ESB -- in whatever form or flavor it's delivered -- should provide a list of basic "services" ... or functions. A brief description of those services is offered on the chart. This is not an industry standard list, but there does seem to be a consensus forming around the basics offered here. We'll see how some of those things are implemented in the IBM products we'll look at later.

### Ability to “Alias” the Service -- Improved Flexibility

Even if we don’t do fancy message transformation, simple routing through ESB provides the benefit of improving flexibility:

- Service users continue to go to same ESB location
- Definition inside ESB modified to point to new service location
- Change hidden from service users

Admittedly a simple example. And dynamic retrieval of an updated WSDL in a Web Services world achieves the same result. But what about application connections that don't use Web Services? Or, what if client has cached copy of WSDL and it's not updated?

Even simple implementations of ESB can serve important role as intermediary that hides service location details behind common entry point. Then it can be expanded as needed to do additional protocols, protocol remapping, message transformation, etc.

More complex examples ...

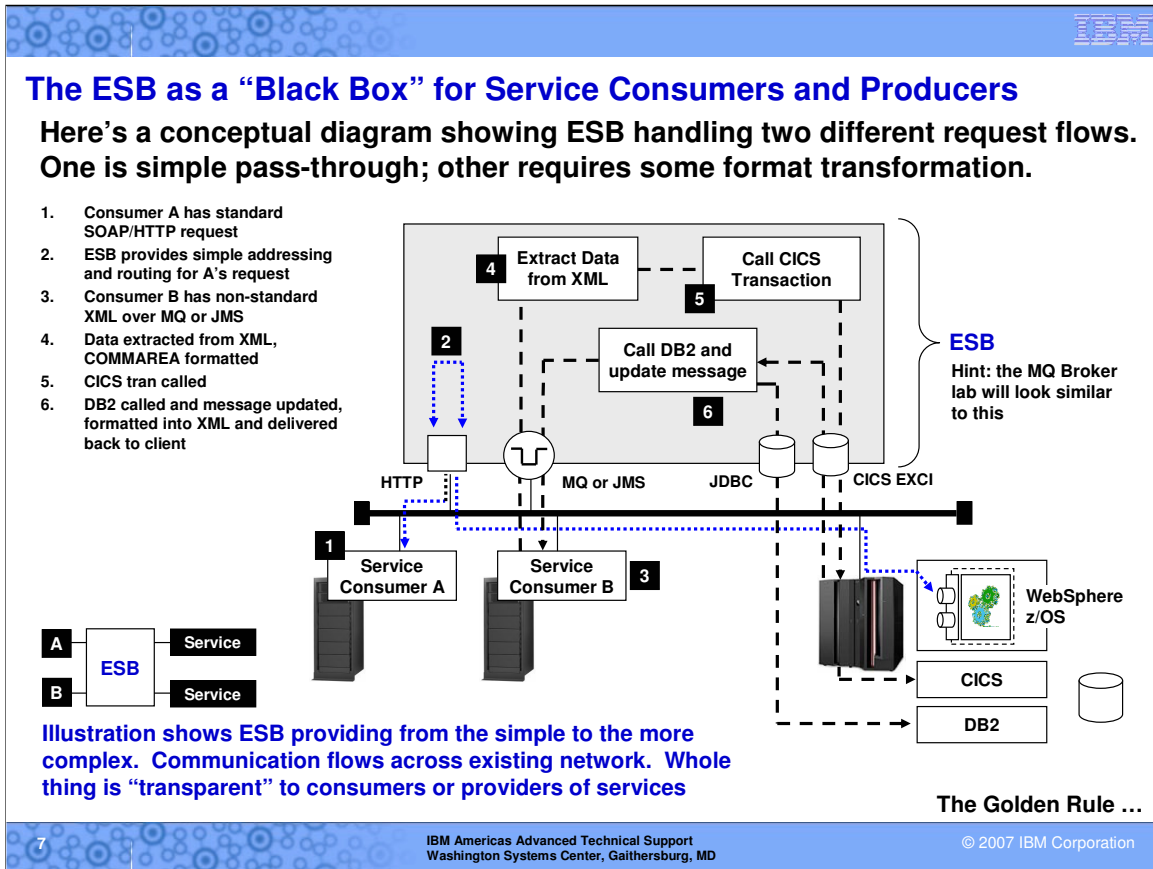
6 IBM Americas Advanced Technical Support Washington Systems Center, Gaithersburg, MD © 2007 IBM Corporation

The previous page showed an example of two flows: one very simple, one more complex. That chart is useful, but it may leave someone with the impression that an ESB is only useful when doing complex message mediation. To help correct that, consider the chart above. It shows an ESB that's doing only simply routing of messages. And in this picture we're showing lots of requesters going after the service.

Now what happens if you have to move the service to a new machine ... perhaps at a new host name, or IP address, or port? That may involve having to go out to all the service requesters and modifying them to point to the new location. But if the request goes to the ESB first, then the definition of the service endpoint can be updated in the ESB. The users are unaware of the change ... their definition of the service location stays the same -- the ESB. But the requests now flow to the new location.

There are other ways to achieve this result, of course. The chart mentions dynamic refresh of WSDL in a Web Services environment. Or a front-end IP proxy device could also do forwarding. So when the example is too simple then the value of an ESB gets a bit obscured. What an ESB provides is this simple “alias” benefit as well as the opportunity to expand later and do message mediation (changing the message in some way); mapping the request to a different outbound protocol; enhancing the request to actually call two backend services rather than just one, even though the user is still issuing one request. The possibilities are quite large.

Let's look at a slightly more complex example.



The concept of an ESB has been described by some as a “patch panel” between requesters and providers. Another way to look at this is for the ESB to be a “black box” connecting requester to provider, with neither really knowing what’s going on inside the ESB. Here’s a picture that strives to make the ESB a little less mysterious.

We’ll walk through this example based on the numbered blocks:

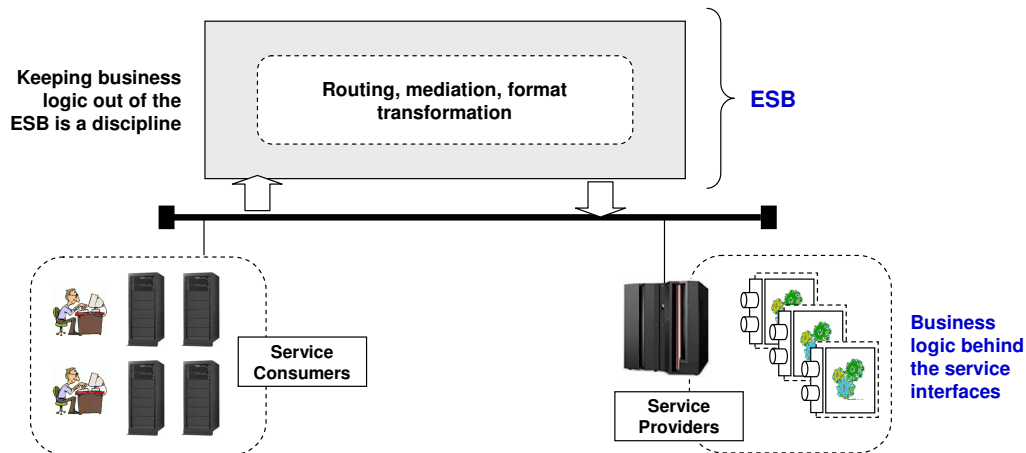
1. Let’s assume we have a service consumer (“A”) who’s method of access is SOAP over HTTP. They seek to use a Web Service hosted in WebSphere for z/OS. Rather than code the WebSphere host and port as the destination, the Web Service client points to the host and port being listened on by the product implementing the ESB.
2. Consumer A’s request goes into the ESB. But for this request there’s no need for any message transformation or protocol switching, so the ESB performs simple routing and sends the request on its way to WebSphere. The response returns to consumer A.
3. Assume you have another service consumer (“B”) who uses a non-SOAP message over MQ or JMS. They seek to access CICS. They too initially connect to the ESB via a queue monitored by the product that implements the ESB.
4. A piece of logic in the ESB decomposes the XML and formats up the COMMAREA
5. It executes a CICS transaction over EXCI
6. It then queries DB2 over JDBC to retrieve a final bit of information, then recomposes the XML and returns it to consumer B

Neither consumer has any idea what’s going on inside the ESB. To them the ESB is simply a “black box” -- a place where each connects to for its services.

The presence of computation going on inside the ESB may appear odd. Does that mean we have yet another place where our applications can run? Let’s look at the golden rule of ESBs ...

## No Business Logic in the ESB!

The ESB has within it a computational environment. It would be *possible* to code business logic there, but it is strongly recommended you do not:



**Why? Because business logic in the ESB starts to break down the “service oriented” approach. It’s a trend back in the direction of tightly integrated and inflexible.**

IBM Product Implementations ...

8

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

The ESB is capable of hosting computational logic. In order for the middleware to do the kind of processing expected of an ESB -- message transformation, routing based on content, protocol switching -- it's going to need to be able to run “programs”. Does that mean it's possible to start doing application business logic in the ESB? It is, but it is strongly recommended you do not.

The reason to avoid this is because it's a trend in the direction *back* towards complex and inflexible application architecture design. The idea is to have your business logic contained in the services components and the composite applications that string together the services. There will be “logic” in the ESB, but it should be limited to logic associated with ESB functionality -- routing and transformation.

How can this be insured? By imposing discipline in the way you use and manage your SOA environment. This gets to the question of “governance,” which we'll cover later.



## IBM's Implementation of ESB into Product

Ultimately we need to get to the point where we can point to something and say "There, that's IBM's ESB on z/OS." Here they are:

**WebSphere Enterprise Service Bus (WESB)**

WebSphere Application Server for z/OS

- Built on the proven J2EE platform of WebSphere Application Server
- Focus is standards-based access and J2EE connector support to backend systems

*They can interoperate*

**WebSphere Message Broker (WMB)**

WebSphere MQ for z/OS

- Built on the proven messaging platform of MQ
- Standards-based access *and* non-standard through a host of connectivity options

**Key Points:**

- *Much* more to cover -- we'll look at each product in more detail in a bit
- Shows how ESB is function implemented in middleware (makes "real" the concept of ESB)
- There is the question of "Which Should I Consider?" which we'll cover next  
*It's going to come down to function needed and degree of non-standards access. WMB is referred to as "Advanced ESB" because of its capabilities -- standards-based and non-standard. WESB focuses on industry standard only.*

**Product Positioning ...**

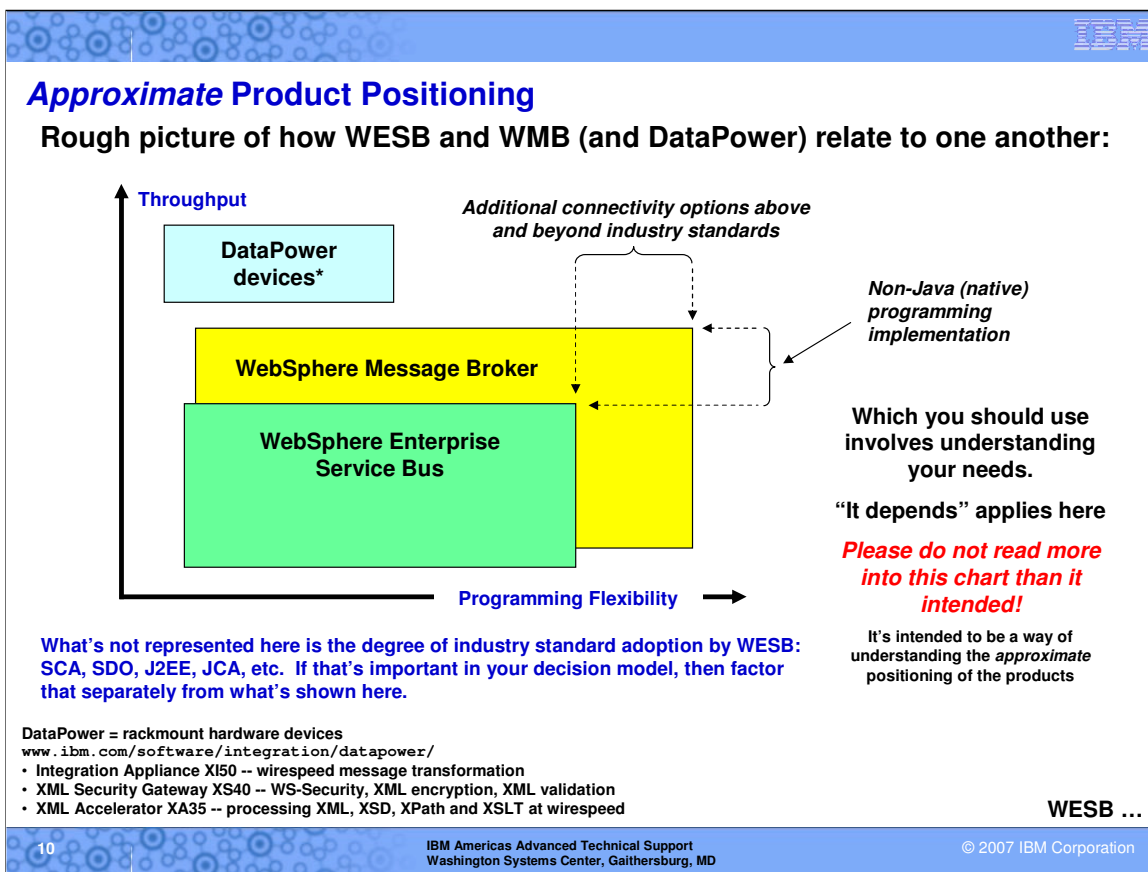
9
IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD
© 2007 IBM Corporation

Let's get practical here and mention the two pieces of IBM middleware that implement the ESB -- WebSphere Enterprise Service Bus (WESB) and WebSphere Message Broker (WMB).

- **WESB** -- This is function that is added to WebSphere Application Server. It provides for message routing and some message "mediation" (which means changes to the message). The focus of this offering is on standards -- J2EE, SOAP, Web Services. It can take advantage of WebSphere's rich array of J2EE connector support to access data.
- **WMB** -- This is function built on the MQ messaging platform. This is something referred to as "Advanced ESB" because it does standards-based message handling (SOAP) as well as non-standard. In fact, the array of non-standard protocols and accessibility of WMB is quite impressive, as you'll see.

We do dozens of charts that go into more detail on these two offerings. So this one chart is intended to be just a brief introduction and nothing more. Still, it's important to understand that there are real products that implement the concepts of ESB we've been talking about. For IBM at this point in time, these are the two ESB offerings.

The natural question comes up -- "Which should I consider?" The positioning of one vs. the other is something best done when you have a handle on what each is capable of. Ultimately it comes down to the capabilities you require, the skills you already have, and the degree of standards compliance you require. The two can interoperate, so the decision really is not "either / or", but rather WESB, WMB or possibly both.



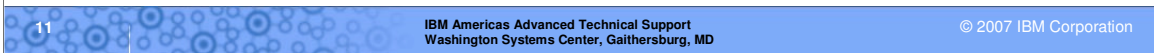

This chart provides a kind of rough positioning of the two products we just mentioned -- WebSphere Enterprise Service Bus (WESB) and WebSphere Message Broker (WMB) -- with a third product family introduced: the DataPower appliances.

**Note:** it is very important that you *not* take this chart to be some kind of formal, precise representation of relative functionality, or relative performance. The chart is really intended to merely provide a visual way of understanding the basic positioning. When evaluating which product is right for you, the familiar phrase "It depends" applies. We don't mean to overuse that phrase, but the truth is it does depend ... on what you're trying to accomplish. Which is why we're presenting this material, so you can get a better understanding of which product does what and therefore start the process of mapping the product capabilities to your needs.

The graph has two axis: throughput and flexibility. That's what's being compared here. I tried to think of a way to represent the extent to which each implements industry standards. That's WESB's strong suit -- industry standard implementations. WMB does some of that as well, but not to the extent WESB does. So if that's an important decision criteria for you, factor that in when considering the different product choices.

We've not mentioned DataPower up to this point. DataPower appliances are rack-mount hardware devices that do some of the same kinds of things you'll see the WESB and WMB products doing. DataPower is a company that IBM acquired a year or two ago. The key to understanding the DataPower concept is to understand that much of their speed and power is due to function being burned into processors specifically designed for the purpose. That provides speed. But at a cost of flexibility. That's why you see the DataPower box positioned to the upper left. We'll not cover DataPower any more in this presentation but the URL is presented at the bottom of the chart and the three available devices are offered in bullet format here with a brief description of each.

DataPower devices can be used in combination with WESB or WMB or both.



ESB, WebSphere ESB and WebSphere Message Broker

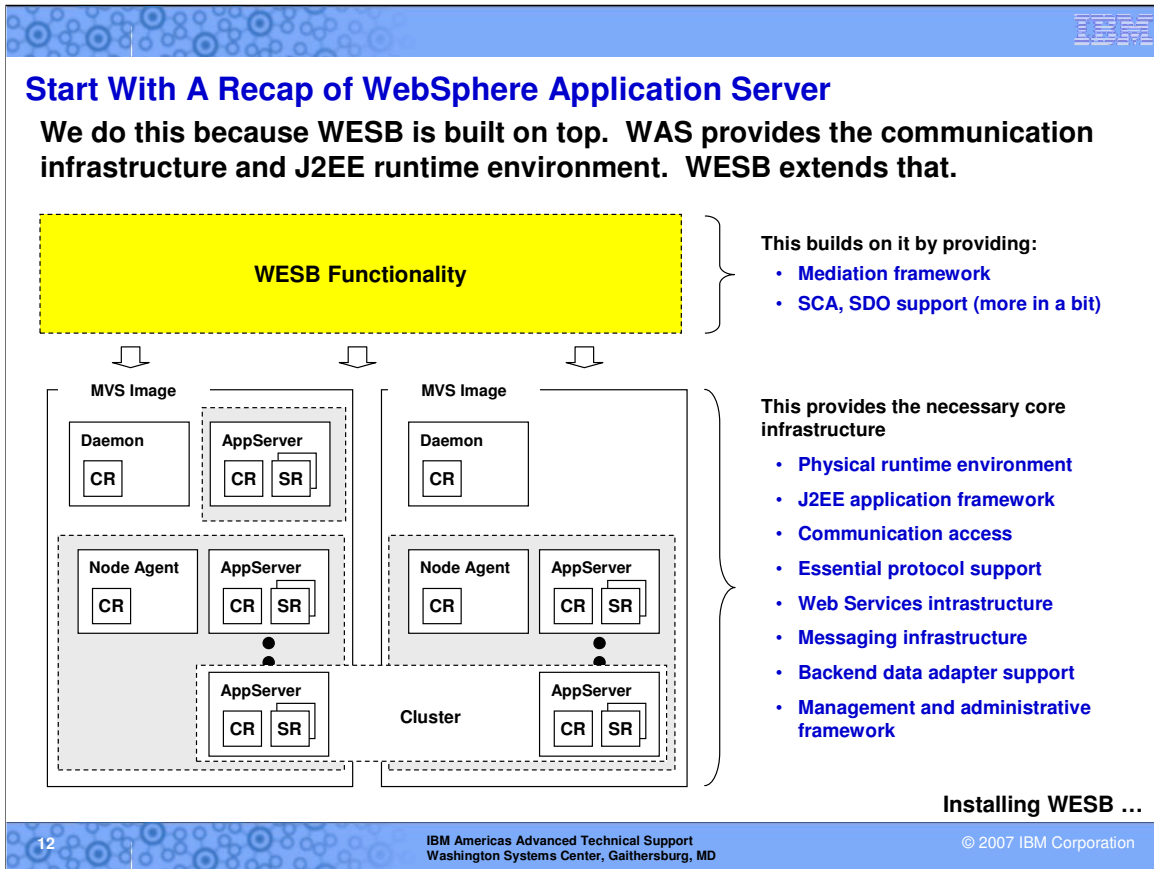
An introduction to

# **WebSphere Enterprise Service Bus (WESB)**

11

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation



To start the discussion of WESB, it's important to begin by reminding ourselves that WebSphere Application Server is the foundation, or base to WebSphere Enterprise Service Bus. What that provides is the necessary core infrastructure, particularly the essential protocol support (HTTP, HTTPS, JMS, RMI/IIOP) and all the backend data connectivity functionality.

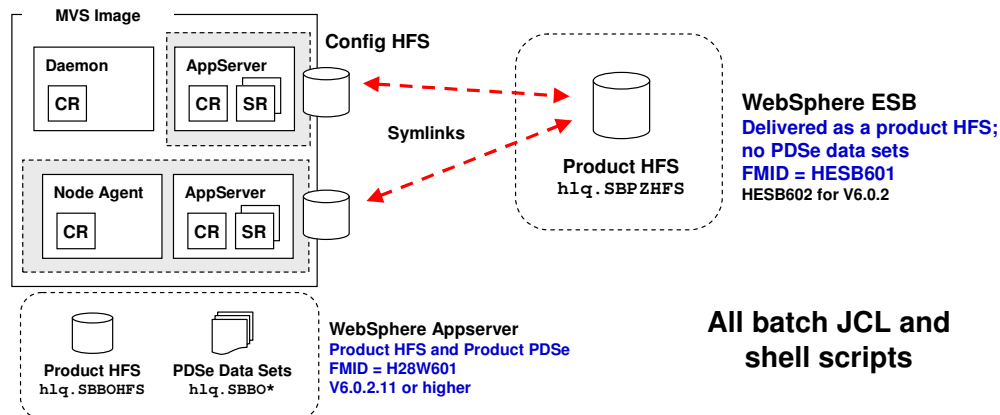
What WESB provides is additional function on top of this infrastructure that provides the key elements of the ESB -- a "mediation framework" and the Service Component Architecture (SCA) and Service Data Object (SDO) support. What those things are will be explained over the next several charts.

We mentioned before how WebSphere Application Server is increasingly being used as a foundation for additional functional support ... this is an example of that.

## How WESB Is Installed

It is a process of linking the WebSphere configuration HFS to the WESB HFS, then “augmenting the profile” (changing it) to include the new function.

There’s also a step to create the database (Cloudscape or DB2)



**Key is this is an addition of function to an existing configuration. Function is introduced with shell scripts that link WAS configuration HFS to the WESB HFS, along with an update of the WAS profile.**

Message Handling Inside WESB ...

13

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

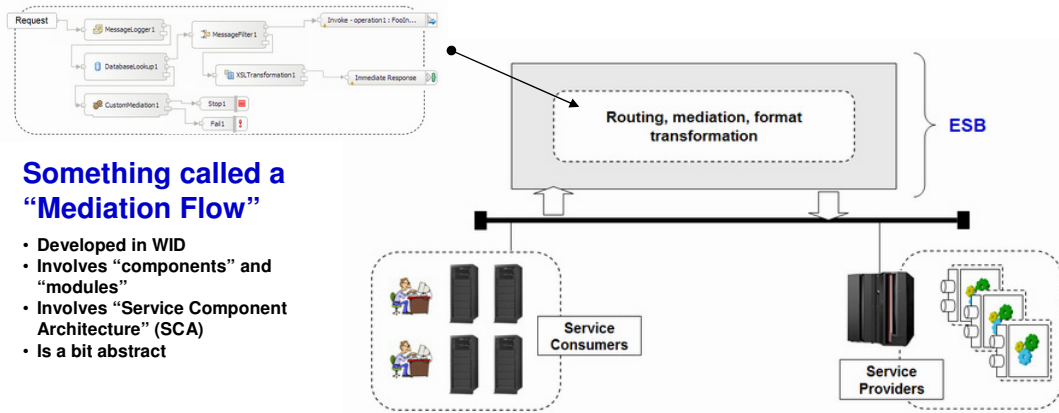
© 2007 IBM Corporation

WebSphere Enterprise Service Bus is a separate product from WebSphere Application Server. The process for installing it involves two basic steps:

- Performing the SMP/E installation of the WESB product itself. This is going to create a product HFS among other artifacts. (But *no* PDSe module libraries. WESB runs entirely within WebSphere Application Server which has its own PDSe module libraries.)
- The “linking” of the WebSphere configuration to the WESB product HFS. This is done with a series of shell scripts that creates symbolic links out of the configuration HFS into the WESB product HFS and “augmenting” (updating) the WAS configurations with several new resources and applications. It’s not hard ... but it does involve carefully running a series of scripts.
- Finally, the creation of the backing database structure. WESB needs to store information about elements of its configuration -- SIB information, deployed mediation modules, etc. -- and it uses a relational datastore for that. Database creation scripts are provided for Cloudscape (a file based relational store that has a JDBC interface; simple, used for initial construction and testing purposes) and DB2 (more robust, used for production implementations).

## Message Handling Intelligence in the ESB

To discuss WESB it quickly becomes necessary to talk about the programming capabilities WESB supports. That will be our focus over next several charts.



It’s a key part of the WESB story. Talking about it requires us to explore how the mediation flow is built and what it can do. It’s a more abstract discussion.

Mediation Framework ...

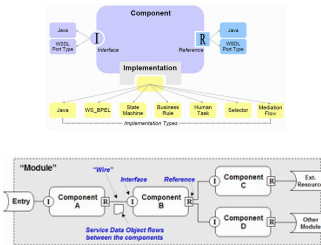
In discussing the WESB we could focus on just the installation and configuration tasks of the product, but that would leave much of the story left untold. We ultimately need to discuss how the intelligence inside the ESB is developed, and *that* requires we talk about the programming elements of the product. For WESB, that involves talking about the “Mediation Framework” and something called Service Component Architecture (SCA). It’s a somewhat complex and abstract topic, but talking about it is necessary to get the story of WESB across.

## Background and History of Service Component Architecture

Launched around 2005, this is intended to standardize development of applications that conform to SOA principles.

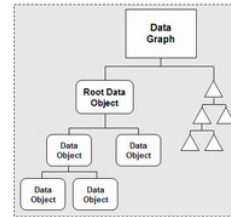
Key Vendors: BEA Systems, Cape Clear Software, **IBM**, Interface21, IONA Technologies PLC, Oracle, Primeton Technologies Ltd, Progress Software, Red Hat Inc., Rogue Wave Software, SAP AG, Siebel Systems, Software AG, Sun Microsystems, Sybase, TIBCO Software Inc.

### Abstract Program Representation



- Orients programming model around concept of services with interfaces and references
- Graphical development tools will assist in drawing out and generating the code

### Abstract Data Representation

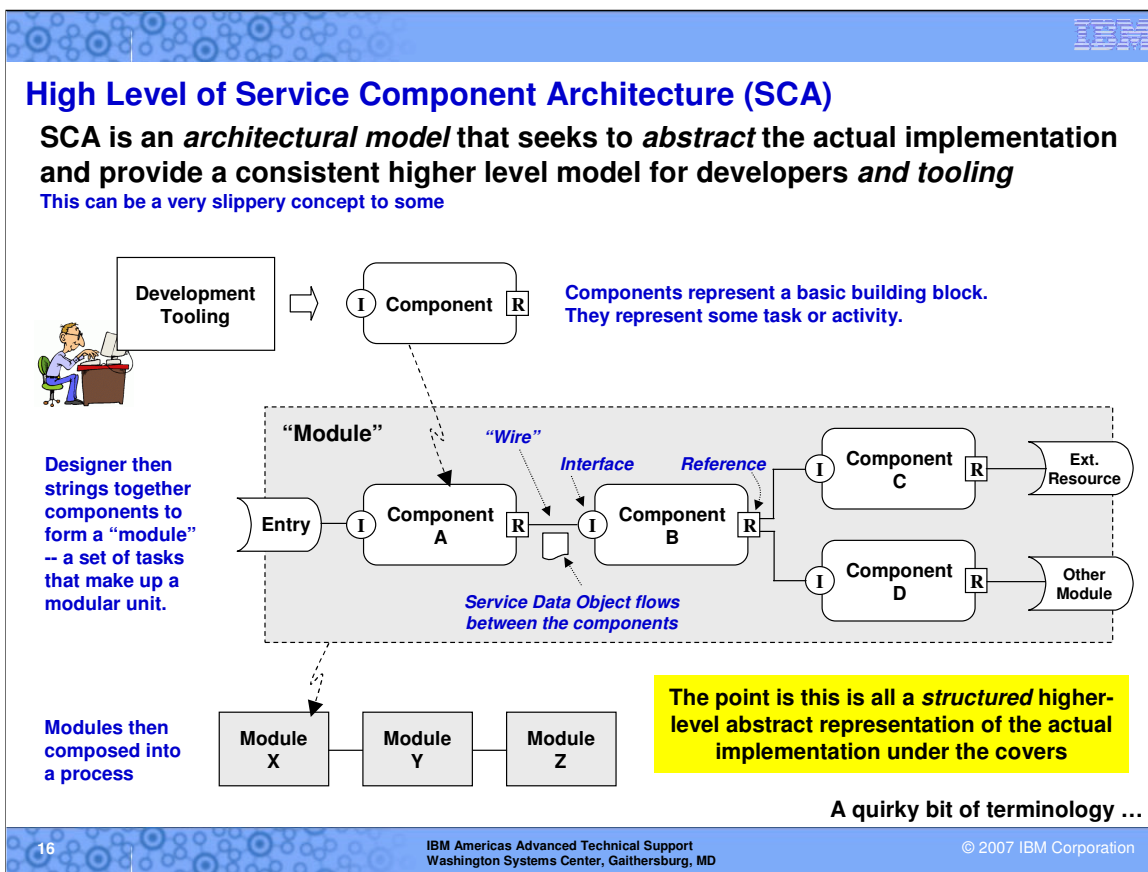


- A standardized way to represent data
- Provides for easier data interchange
  - Within a mediation flow
  - Between service implementations

WESB's programming model is based on SCA, so discussing it requires we touch on SCA.

High Level of SCA ...

text ...



Service Component Architecture -- SCA -- can be a somewhat slippery thing to get your mind around at first. Ultimately it's an development **architecture** ... and what we see in the picture above is a representation of some essential elements of this architecture. But when all is said and done what results are *programs that are deployed to a runtime and executed*.

If we keep that in mind -- that SCA is a way to use abstract symbols to represent real code -- it may be easier to see what this is all about.

We start out by by defining a "component." A component represents some task or activity. A developer sitting at a developer tool (WebSphere Integration Developer -- WID) uses the graphical environment of that tool to draw out a "component," then he or she defines the properties and settings for that component. The tool is responsible for generating the actual code that implements the functions of the component.

A component has an input (an "Interface") and an output (more properly called a "Reference"). In other words, the task represented by the component has an Interface that defines how the task is invoked and what requirements it has. We'll see more detail on what interfaces WESB supports. The component has a Reference to whatever takes place after the component's task is done. The connection between the component and the next task (component) in the chain is called a "wire." A developer composes a flow of activities (tasks, called "components"), wires them together (draws the lines that represents the connections) and sets the various properties and settings for everything. Multiple components then make up a "module," which is a functional unit of work in a business. Multiple modules can be strung together to form a "process." (The classic example is the insurance claim process, which involves many individual tasks. Each task is a component. Multiple components make up the process.)

The final point -- data flows between components. This data is represented in a structured way, and it's called the Service Data Object (SDO). We have more to talk about this as well.

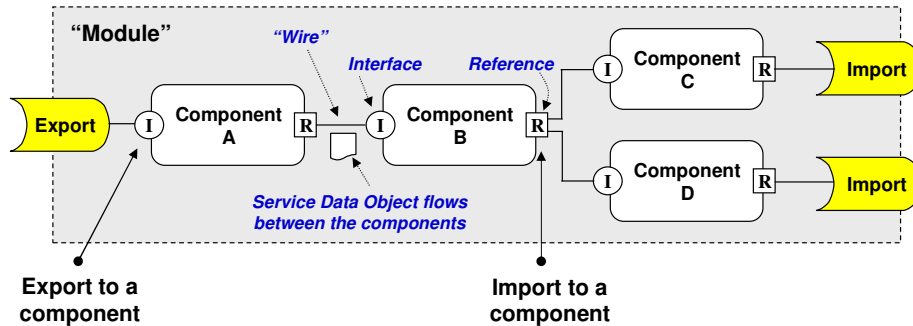
The highlighted box in the lower right of the chart is the key point. Try to hold onto this as we go forward and see more details of this stuff.

*Ultimately this gets generated into code that is run in the WebSphere Enterprise Service Bus.*



## A Quirky Bit of Terminology

The SCA terminology has something that seems at first to be a bit backwards. The input to a module is called the “export”, the output the “import” ...



We bring this up so that you're aware of the terminology. Much of the SCA documentation references “Export” and “Import” and it's important you understand what's being referred to.

Service Data Objects (SDO) ...

17

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

If you spend any time at all reading the documentation about SCA, you'll come across the term “Import” and “Export” relative to components. And initially you may be confused, because the terms are used exactly backwards from what you would think. The flow through the Interface (the front end) of a component is called “Export” and the flow through the Reference (the backend) is called “Import.”

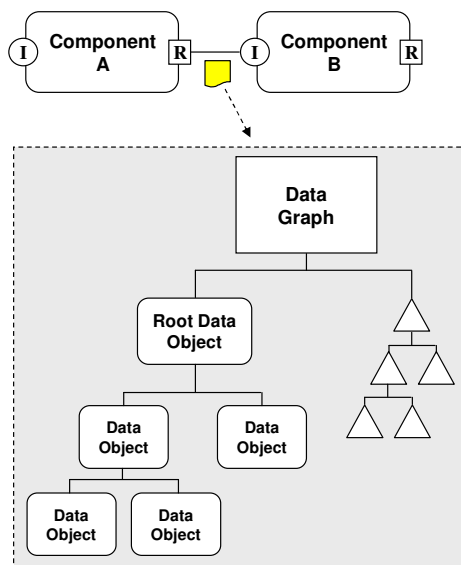
There is probably a good reason for this. The author of this presentation is uncertain what this reason is. The important thing for us is to simply understand that this terminology is being used, and to accept it. There are many more important things to understand and it may not be worth the effort to spend time trying to really comprehend the precise use of this terminology.

Just know it's being used and let's move on.

## Service Data Objects

A structured way of representing data handled by components *and* data that flows between components. When flowing, SDO's are in XML format.

This is often represented in a "tree" format:



### Data Graph

A "container" (outer wrapper) for the data objects that are held within. A message flowing will typically have more than one data object.

### Data Objects

A data object represents a piece of data. A person's name, an invoice number, a price, whatever. Multiple data objects are typically part of a larger message. They too are arranged in a tree format which represents their relationship to one another

Each data object has:

- Name of data object
- Type of data object (simple/complex; scalar/array)
- Value (as well as the default value)

### Change Summary

Represents the incremental changes made to the data objects as the SDO moves between components.

Messages are a type of Data Object ...

As we mentioned, data is going to flow between the components of a business process flow. If it didn't there'd be little reason to bother with this stuff -- moving data, using data, and modifying data is what this is all about.

The designers of this SCA thing knew that if data was unstructured -- just a blob of characters -- then handling the data would be very cumbersome and difficult. It would be necessary to parse out pieces of it, and the rules for parsing would depend on the nature of the data itself. "The address starts at character 29 and runs 12 characters." It would be a mess.

So they came up with a concept for representing data as it flowed between components. And it's all based on the notion that almost all data has a logically structured nature to it. A list of customers can be logically arranged -- the first customer represented by their unique reference, and then under that the elements of information related to *that* customer; the next customer represented by their unique reference, and under that *their* information.

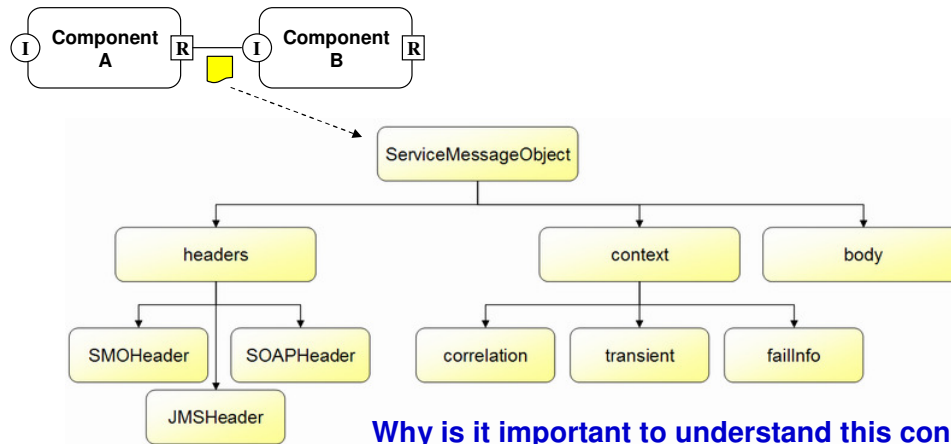
With that we can describe the Service Data Object (SDO). It consists of:

- **Data Graph** -- which is really like an outer envelope. The Data Graph is information about what's held within the envelope.
- **Data Objects** -- these are the units of information held in the SDO. As we mentioned, this is arranged logically, based on the nature of the information. The logical representation is that of a tree structure. The "Root" may be something like "Customers," with the next data object layer down the unique identifier of each customer held in the SDO. Under each of those comes the specific information about each customer.
- **Change Summary** -- a way to hold information about how information within the SDO has been changed as this SDO flows between the components of a process flow.

Understanding that data can be arranged in a logical structure like this is important because messages are a form of data, and accessing pieces of information within a message is *much easier when the data is represented logically like this*.

## Service Message Objects

Are a form of Service Data Object. The contents of the message is represented in a tree format:



### Why is it important to understand this concept?

Because any transformation of the message is going to be made against this structured format. Querying specific data elements will be aided by such a structured format. All with structured change data.

We'll see something very similar with WebSphere Message Broker

Let's look at the "Component" ...

19

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

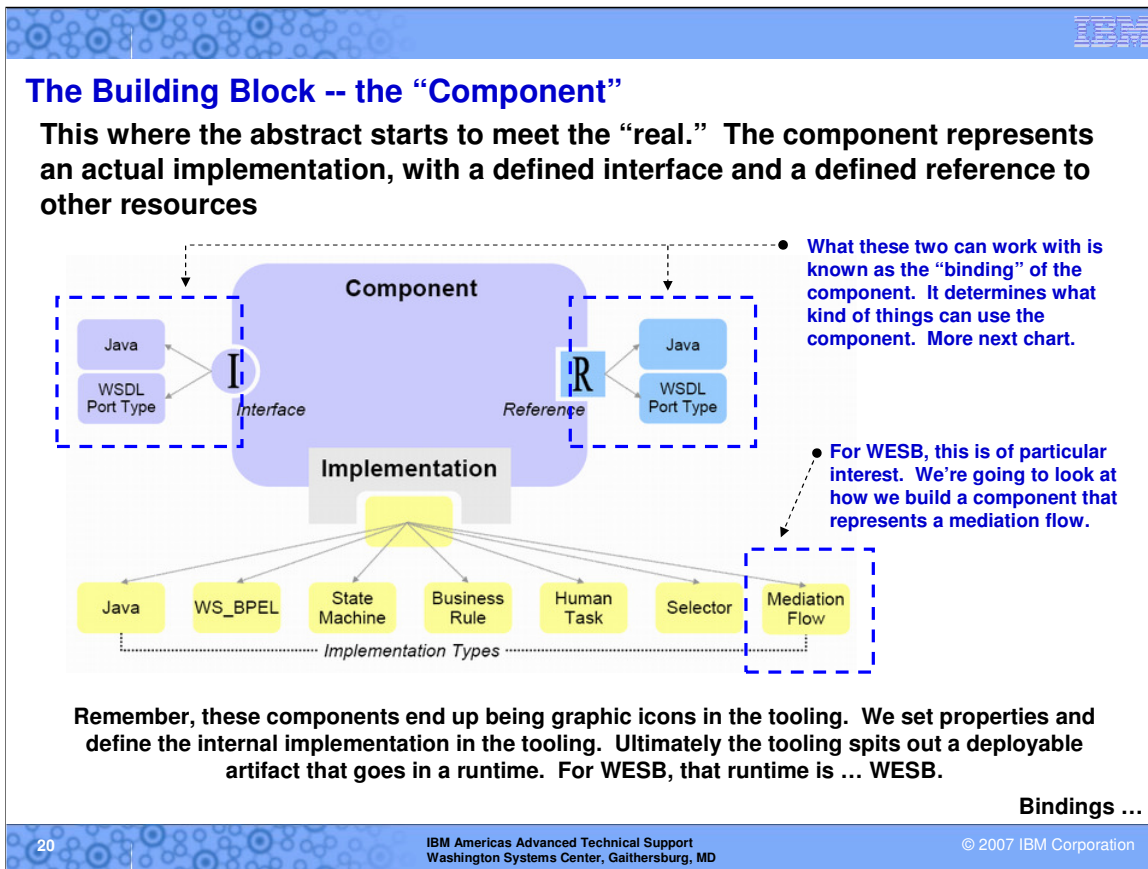
Messages are a form of data objects, and because of that we can represent the data within a message in a logical, structured way as well. The term "Service Message Object" -- SMO -- is used.

Here's why this is important -- if data is represented in a logical tree structure like this, we can access elements of the data by reference the element in a consistent way. For example, look at the picture above. Let's say for some reason we want to access the SOAPHeader. In our program we could reference that with something like this:

```
myVar = ServiceMessageObject.headers.SOAPHeader
```

That's possible because the data is held in a logical tree structure. Were it just a blob of characters, then we'd have to "substring" (or parse) the data out, but only if we knew where it started and where it ended.

When we get into the WebSphere Message Broker portion of this presentation we're going to see something called a "message tree." Even though WMB does not adhere to the SCA standards, it is using *similar concepts*. A WMB "message tree" is a logical, structured representation of the message.



Now we can take a closer look at the "component" and start to see some of the specifics of this that relate to our previous discussions of SOA, Web Services and WebSphere Enterprise Service Bus.

- The Interface of a component defines what outside the component may reference it. This is known as the "binding." SCA is built around the Web Services and J2EE environment, so there are two basic things that can reference the SCA component -- a Java program or a Web Service.
- The Reference of a component defines what it will go to to get other information. This too is known as a "binding," though it's the reference binding while the other is the interface binding. We see the picture above showing Java and WSDL.

**Note:** we'll see on the next chart that the list of what may interface or reference is a bit longer than just Java or Web Services.

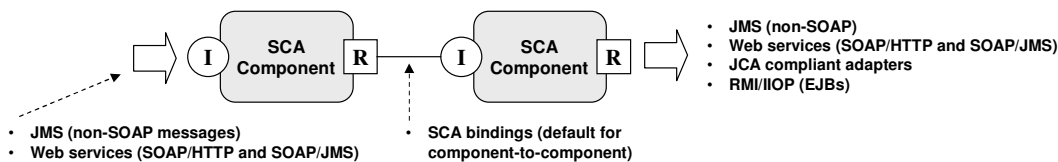
- The Implementation is a logical representation of what actually happens when a component is invoked. The list of things across the bottom shows the various programming languages and other things -- *notice that they're not all programming languages* -- that may be behind a component. A component may well be a "human task" -- that is, a part of the process flow where someone needs to get involved to do a manual task. For the sake of our WESB discussion our focus is going to be on the right-most box -- the "Mediation Flow"

A "mediation flow" is one implementation of an SCA component. For WESB, we're about to see that WebSphere Integration Developer (WID) provides a series of "mediation primitives" -- sub-components, if you will -- that can be used to construct the internal workings of a mediation flow component.

But next, let's look at the bindings of a mediation component.

## The “Bindings” of an SCA Component

These define what can invoke a component, and what a component can invoke. It's not “anything” -- there's a defined set of things:



## These are all open standard protocols

(WMB has these *and* non-standard interfaces)

Development tool ...

21

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

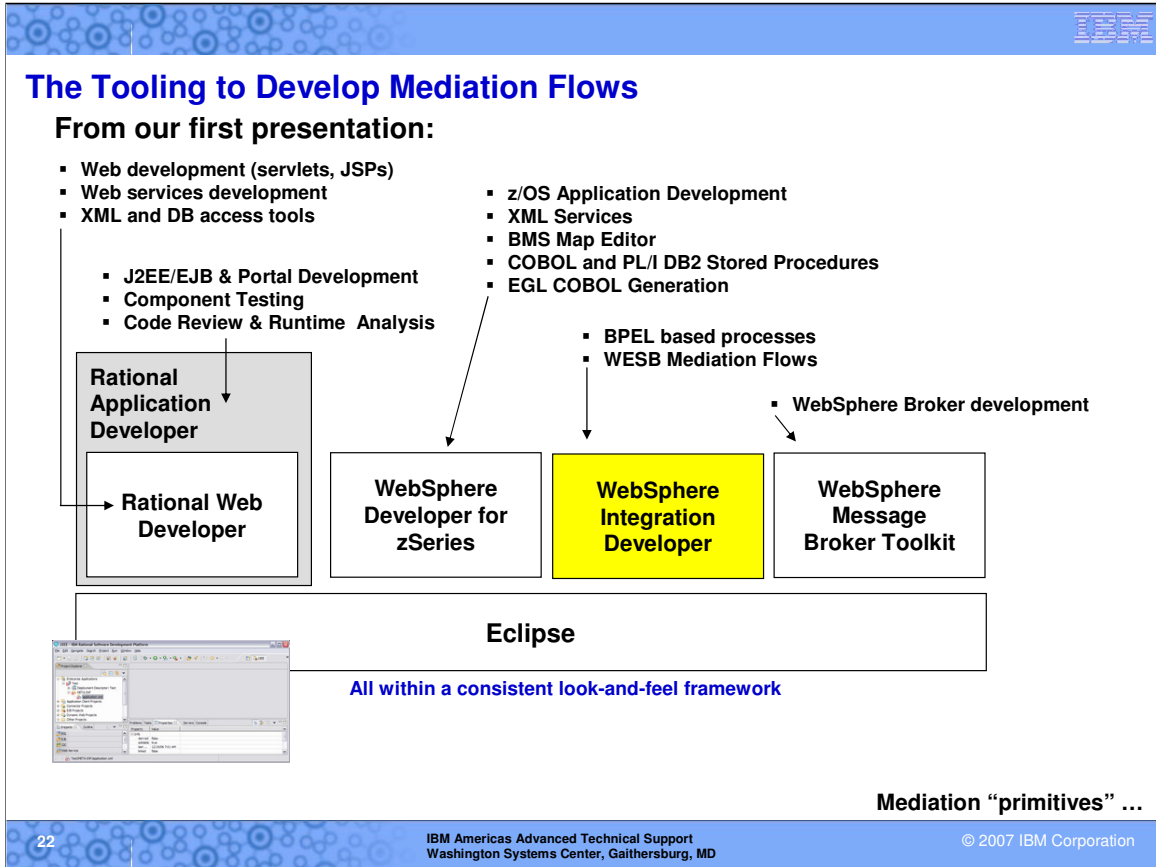
© 2007 IBM Corporation

The “bindings” of an SCA component define what can be used to invoke the component, and what the component can in turn invoke. The top-half picture shows the bindings possible, and the bottom half shows essentially the same information, but formatted in a way that brings the picture closer to home with respect to WebSphere Enterprise Service Bus:

- Service consumers can connect to and invoke a mediation flow inside of WESB using either a message over JMS (not necessarily SOAP -- could be another format), or a Web Service request in the form of SOAP over HTTP or SOAP over JMS.
- On the “back side” of the WESB mediation flow, where the service providers are, you have more options: the same as input (JMS message, or Web Service over HTTP or JMS), as well as invoking an EJB via RMI, or any of the JCA adapters WebSphere supports to access backend data like CICS or IMS, or any relational database over JDBC, or another SCA component.

Here's one of the first places where we can start to position WESB and WMB. WMB permits Web Services and JMS, as well as a good deal of other non-standard connections. In that sense, WMB is often referred to as an “advanced ESB” because of the additional connectivity options.

To build a WESB mediation flow, we use the WebSphere Integration Developer product (WID) and construct the component using what are known as “mediation primitives,” which can be thought of as a sort of “sub-component”.

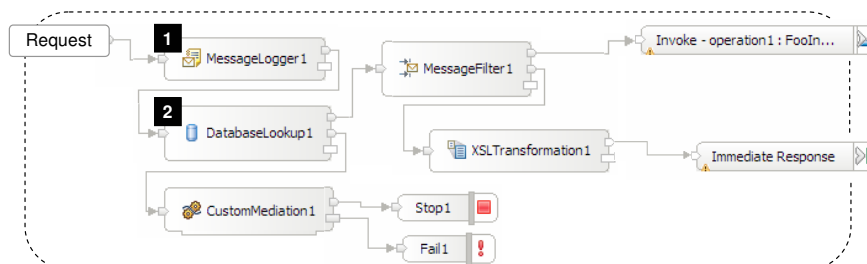


We'll cycle back quickly to our picture of the tooling architecture and see that the component that builds the WESB mediation flows is WebSphere Integration Developer (WID), which is another functional add-on to the Eclipse base.

We are about to launch into a bit of a discussion about the programming architecture used by WESB. It's important to get that understood because it's the basis for much of the workings of WESB.

## Mediation “Primitives”

“Primitives” are like “sub-components” -- functions you can use in constructing your WESB mediation component.



**1**

Allows logging of whole or part of SMO to a database table

Original message propagated through the output terminal

Schema of database is fixed

- Timestamp
- MessageID
- ModuleName
- MediationName
- Message
- Version

**2**

Augments message with information from a database

Obtains key value from message

Adds data from matched database row into message

Actions:

- Output terminal fired if key is found in DB
- KeyNotFound terminal fired if key not matched
- Fail terminal fired if an exception occurs during processing

Message Filter and XSLT ...

We're going to show several charts of “mediation primitives,” and we're going to do it with a hypothetical mediation flow as shown above. We'll then walk through each primitive and explain what it can do.

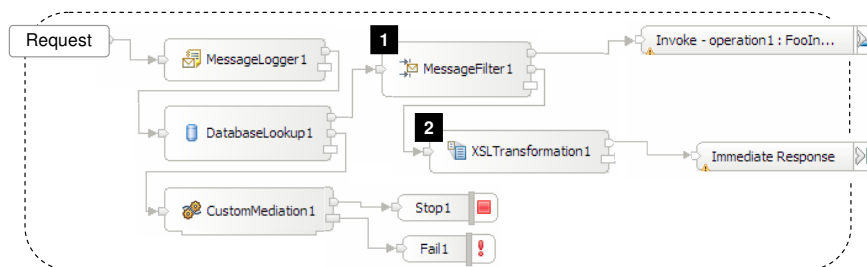
**Important Note:** you construct a mediation flow using the primitives you need. We are definitely *not* saying you must use all these in each flow you construct.

- The *MessageLogger* primitive is used to log some portion of the Service Message Object (the message received) received message to a database. The database schema is fixed, but it does give you the flexibility to log all or a part of it.
- The *DatabaseLookup* primitive is used to augment a message with information retrieved from a database. You specify a value from the message to be used as a key lookup in the database. The data from the database row that matched the key is then added to the message. This is not intended to be used for complex database queries ... this is a more simple key search with the entire row returned. If the key is not found, then the “KeyNotFound” terminal (output) is where processing goes.



## More Mediation “Primitives”

### Message Filter and XSLT



**1**

Allows messages to be routed along different paths depending on contents

Has a dynamic number of output terminals

Allows filters to be specified which determine

- An XPath pattern to match in the message
- Which terminal to fire if the pattern matches

Can fire either the first filter to match, or all matching filters

If no filters match the default terminal is fired

If an exception occurs during processing the fail terminal is fired

**2**

Transforms the message from the input terminal type to the output terminal type

May work on the whole SMO, or any part of it (body, context, headers)

Uses the graphical XSLT Mapping Editor to help define the XSLT

May optionally select to perform validation of the incoming message

Output terminal fired on successful transformation

Fail terminal fired if transformation fails

Custom Mediation, Fail, Stop ...

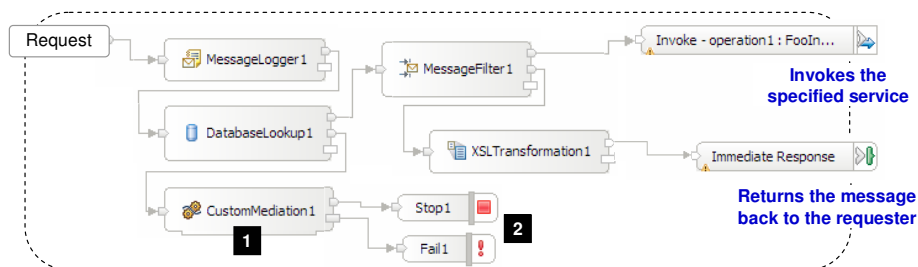
More primitives:

- The *MessageFilter* primitive is used to determine the route a message flows, based on an interrogation of the contents of the message. The “filters” (routing criteria) are based on “XPath,” which is an emerging standard for a structured way to query the contents of an XML file (and Service Message Objects are maintained in XML as they traverse components). The number of “output terminals” is dynamic; that is, you can specify multiple destinations if multiple filters are matched.
- The *XSLTransformation* primitive is used to modify the message, either the whole thing or some portion of it. It uses XSLT to do this ... XSLT stands for “Extensible Stylesheet Language Transformations” and it is a way to transform an XML document to another format, based on a template that describes the before and after format. This primitive would be used when the service requester supplies an XML request document that doesn’t quite match what the service provider expects.



## More Mediation “Primitives”

### Custom Mediation and Fail, Stop



**1**

Enables mediation flows to contain logic not possible with the supplied primitives

Implementation logic may be supplied using:

- An existing SCA component or import
- A Java snippet
- Visual programming

Custom mediations can work on either the body or whole SMO

Always have one input, one output and a fail terminal.

Fail terminal fired if any exception is generated by the implementation

**2**

**Fail:**

- Causes the flow to terminate execution at that point and a `FailFlowException` to be returned
- User may specify the exception error message

**Stop:**

- Causes execution of a particular path of a flow to stop
- Does not terminate whole flow execution
- Leaving a terminal unwired is equivalent to wiring it to a stop primitive
- If wired to a fail terminal will cause the failure to be silently consumed

An Example of a Mediation Flow ...

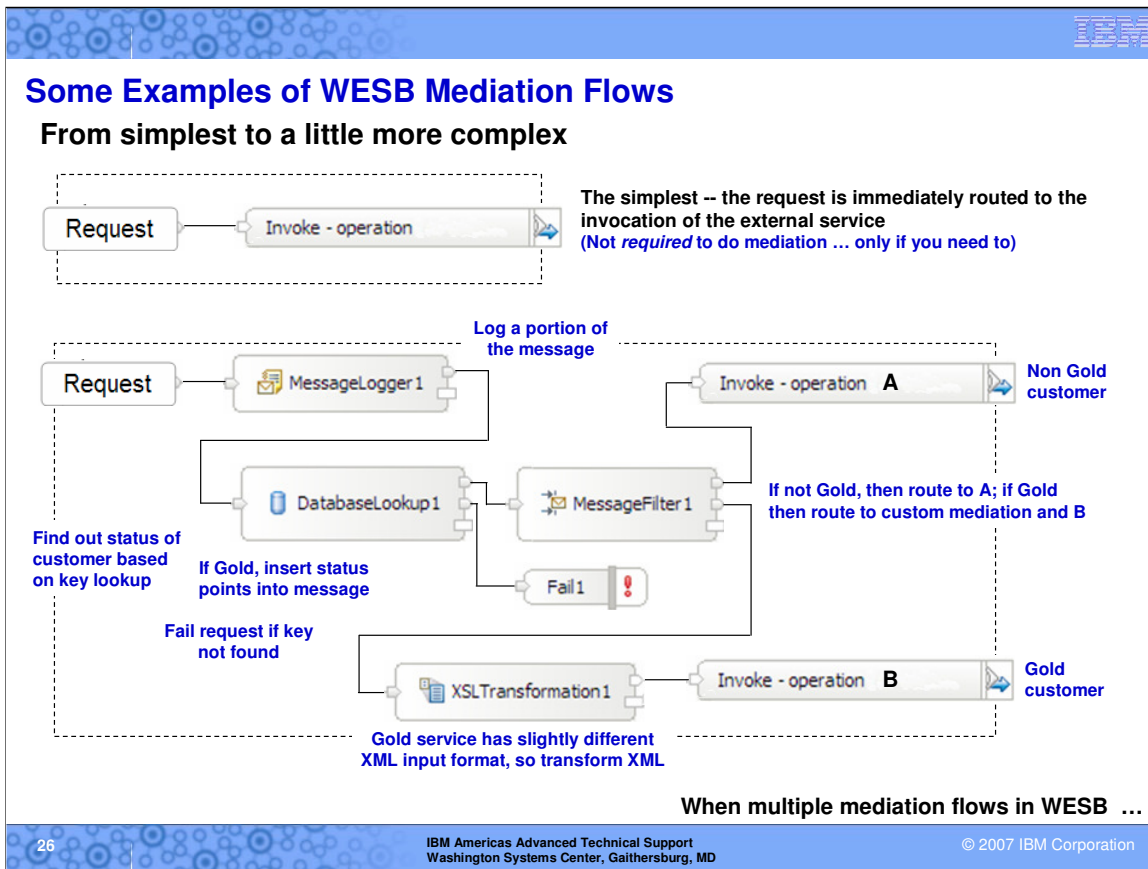
More primitives:

- The *CustomMediation* primitive is a way for you to supply mediation logic that is not otherwise provided in the supplied primitives.

**Note:** we mentioned before that one should never code business logic in the ESB. Here's a case where one could end up doing that if one is not careful.

The implementation logic for the custom mediation can consist of a snippet of Java, code generated by the visual programming element of WID, or can involve the importation of a separate service or SCA component.

- *Stop* and *Fail* -- there are ways to terminate the flow. The primary distinction between the two is that *Stop* halts only that branch of the flow while *Fail* makes the entire flow cease, even if other things are flowing. If you recall, the *MessageFilter* is capable of routing to multiple branches of the flow, so it's possible to have multiple things going on in the flow. *Fail* causes the whole thing to halt; *Stop* causes just the branch to stop.



We we put the puzzle back together with two examples of a mediation flow. In the top portion we show the very simplest example -- a request (which is really the definition of the “interface” to the component) and an “invoke operation,” which could mean a connection to another component, or an invocation of a web service. It’s essentially a “straight-through” operation -- in and out. It illustrates an important point about the ESB ... doing mediation or message transformation is not required. It’s there if you need it, but if you don’t require it you can provide a simple pass-through mechanism like this.

The second example is a bit more complex:

- The request is received and the processing flows to a *MessageLogger* primitive. Some portion of the message is logged to the database.
- The *DatabaseLookup* primitive is used next to look up the status of the customer in the database based a key that’s part of the message. This is to determine if the customer has “gold” status or not. The status is added to the message. If the lookup doesn’t work, the whole flow goes to a *Fail* primitive and an error message flows back.
- The flow proceeds to a *MessageFilter* primitive, where routing is going to be done based on the newly-inserted status of “gold” or “regular”. If a non-gold customer, then the flow is routed to the invocation of an external service A; if gold, then we flow to an XML transformation primitive.
- In the *XSLTransformation* primitive is used to transform the originally received XML into a different format so the gold customer’s service can be invoked properly. Once the transformation has been completed, the flow proceeds to the invocation of that service.

An instance of WESB won’t likely have just one flow defined to it; it’ll have several (perhaps hundreds). So how does WESB know how to correlate a received message with the particular flow to utilize? It’s based on the bindings defined to the interface of the module.

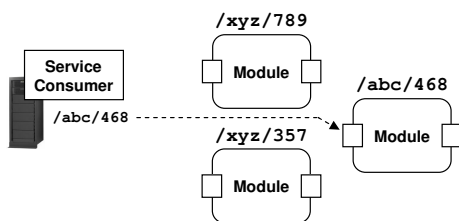
## WESB Hosting Multiple Mediation Flow Modules

It's capable of having many such mediation flows deployed. How then does WESB know *which* flow to invoke upon receiving a message?

It has to do with the “bindings” on the interface of the module. That defines where the module expects to receive input.

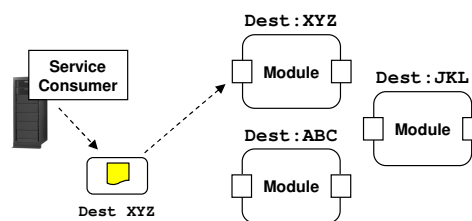
### For modules with SOAP/HTTP bindings:

WESB maintains a table of URIs associated with each module. When a URI is received, WESB invokes the module with the matching binding



### For modules with JMS bindings

WESB maintains a table of JMS destinations (“queues”) associated with each module. When a message is found at a destination, WESB invokes the module with the matching binding

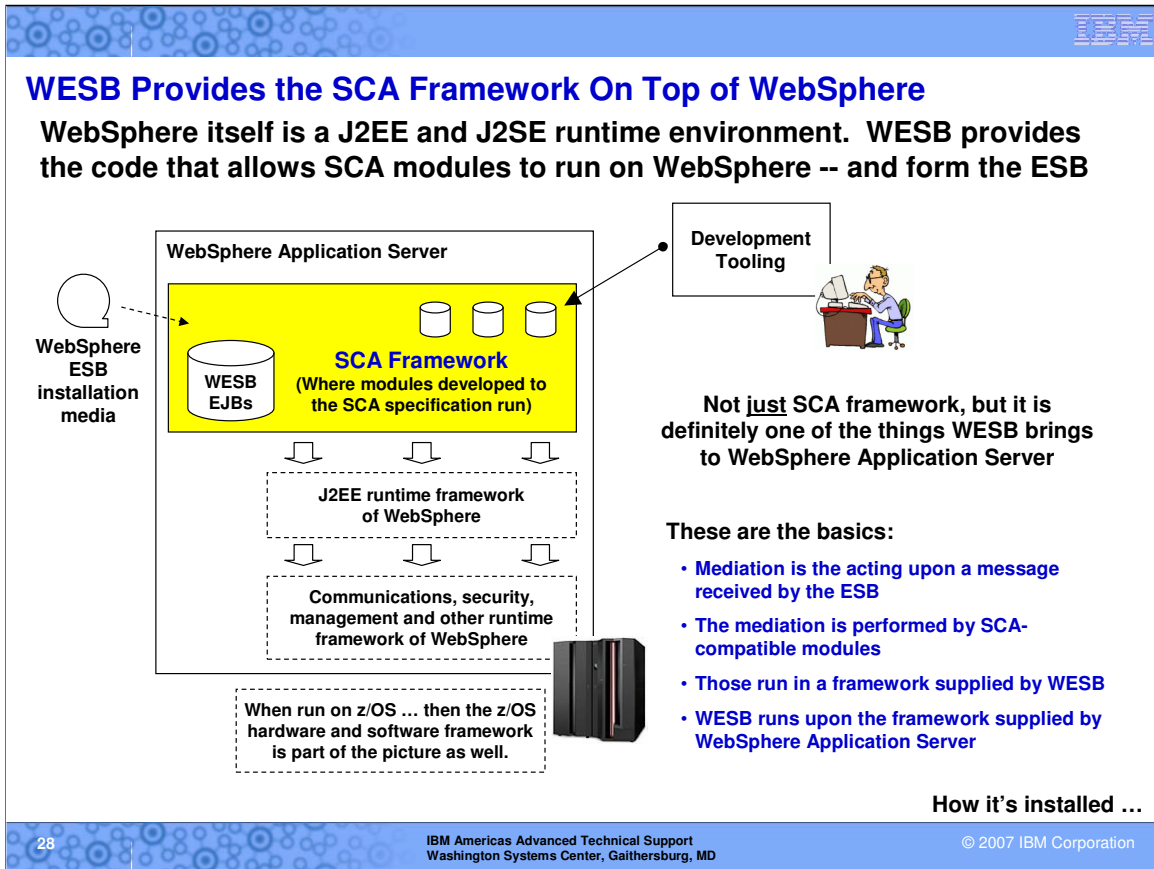


Very similar to what WebSphere Application Server itself does to association inbound requests with application -- webapps, web services, etc.

Cycle back ... the big picture ...

When multiple mediation modules are deployed into WESB, the way it knows how to correlate a received message to a particular module is based on the defined bindings of the interface. In many ways it works like WebSphere itself does when a URL is received. Let's look at this from two perspectives -- for SOAP/HTTP and for SOAP/JMS:

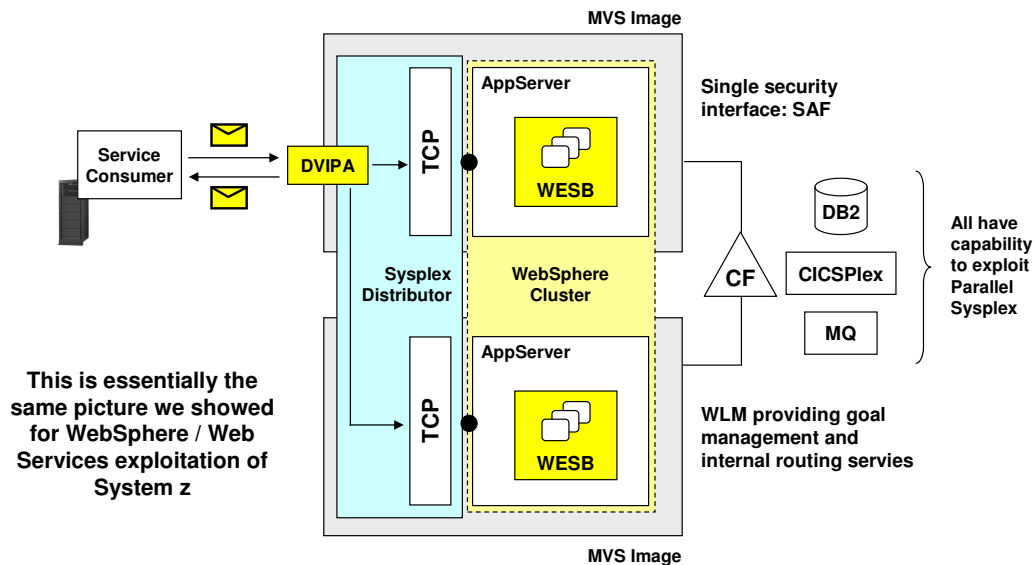
- SOAP over HTTP -- when multiple mediation modules are deployed into WESB, WESB will read the deployment descriptors for the modules and maintain in an internal table a listing of all the URI strings -- context roots -- in its runtime. Then, when a request is received, WESB matches the URI against its listing of deployed modules and if it finds a match it invokes that particular module. As mentioned, this is very similar to how web modules are invoked for standard browser requests in WebSphere Application Server itself.
- SOAP over JMS -- the bindings on modules set up for SOAP over JMS will define a “destination,” which is similar in concept to a queue. WebSphere will monitor the destination and when a message comes in on the destination, and then match it to the modules defined with that destination as part of its binding.



Here again is a big picture representation. What we see is a kind of progressive layering of capabilities, starting with the lower level functions provided by the z/OS software and hardware stack; then building up to the communications, security and other runtime framework elements of WebSphere Application Server itself. WebSphere of course has a fully compliant J2EE runtime environment, which is where the WESB product operates. WESB provides the SCA framework in which the mediation flows are executed. WebSphere Integration Developer is what creates the mediation modules using the graphical environment and the “primitives” we described earlier.

## Exploitation of System z Strengths

Not so much by WESB directly, but by virtue of WebSphere Application Server, which is the runtime framework on which WESB operates:

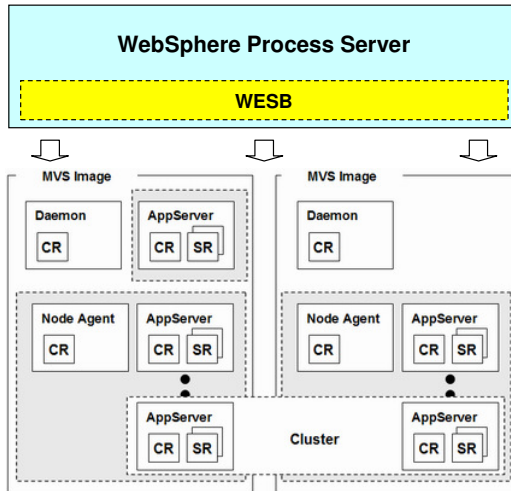


A peek at WebSphere Process Server ...

WESB is essentially a special-purpose application that runs inside of WebSphere Application Server. WebSphere Application Server is capable of taking advantage of the underlying strengths of the z/OS platform, through such things as WebSphere clustering across multiple LPARs in a Parallel Sysplex. Sysplex Distributor and DVIPA out front can be used to route work to the copies of WESB running in the cluster members, and behind the scenes the shared capabilities of Parallel Sysplex for things like DB2 data sharing, or CICS or MQ, can be taken advantage of. And of course underlying this whole picture is WLM, which is managing the workload of it all.

## A Peek At What's Coming Later

Later we'll look at **WebSphere Process Server (WPS)**, which it turns out has **WESB** inside of it. WPS provides full "Process Choreography"



"Process Choreography" is the act of marshalling a more complex set of tasks -- SCA modules and other business tasks -- into a business flow.

This is the tying together of the reusable services -- I/T services and human task services -- into a "choreographed" ("defined", "controlled", "organized") flow.

WPS is the engine that does this. It makes extensive use of WESB inside of itself as well as the SIB of the underlying WebSphere.

Reference information ...

Here we'll offer you a peek at what's coming later when we talk about WebSphere Process Server (WPS). That product, it turns out, has WESB folded in under the covers. Install WPS and you've got WESB. Why? Because you can think of WPS as a super-set of WESB. WPS brings to the table all that WESB does, as well as the ability to compose higher level business processes that are comprised of different tasks. WPS "choreographs" (coordinates the flow) of those processes.

31

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

## Reference Information

### WebSphere Enterprise Service Bus

<http://www.ibm.com/software/integration/wsesb/>

### Service Component Architecture

<http://www.ibm.com/developerworks/library/specification/ws-sca/>

### Service Data Objects

<http://www.ibm.com/developerworks/java/library/j-sdo/>

### Redbook



<http://www.redbooks.ibm.com/SG24-7212>

**Has a very good overview of SCA and SOA as well**

**Next: WebSphere Message Broker ...**

Reference information for WESB.



ESB, WebSphere ESB and WebSphere Message Broker

An introduction to

# WebSphere Message Broker (WMB)

32

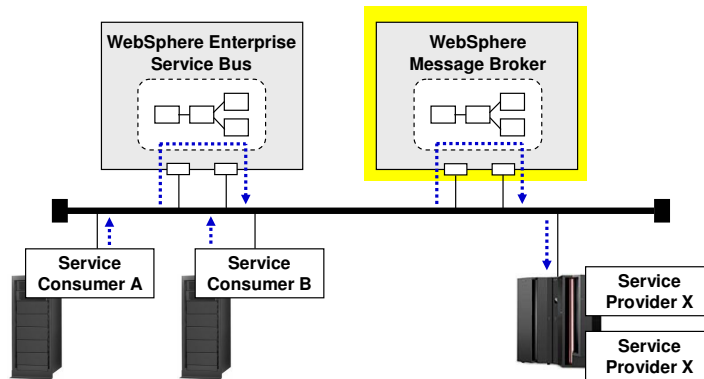
IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation



## The Highest of Overview Pictures

At the *high conceptual level*, WebSphere Message Broker (WMB) is a lot like WebSphere Enterprise Service Bus:



- WESB built on WebSphere Application Server
- WMB built on WebSphere MQ

- Both are IBM product implementations of the ESB concept
- Both host “flow programs” that determine where requests go and if any changes are made to them as they travel through ESB
- Both support a range of ways to connect to them
- Both are implemented on z/OS as a series of started tasks and address spaces
- Both employ Eclipse-based tools to develop their flows
- They can communicate between each other.
- WPS can choreograph processes in either

Let’s look at what WMB does before we get into too many “positioning” questions

What WMB is ...

33

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

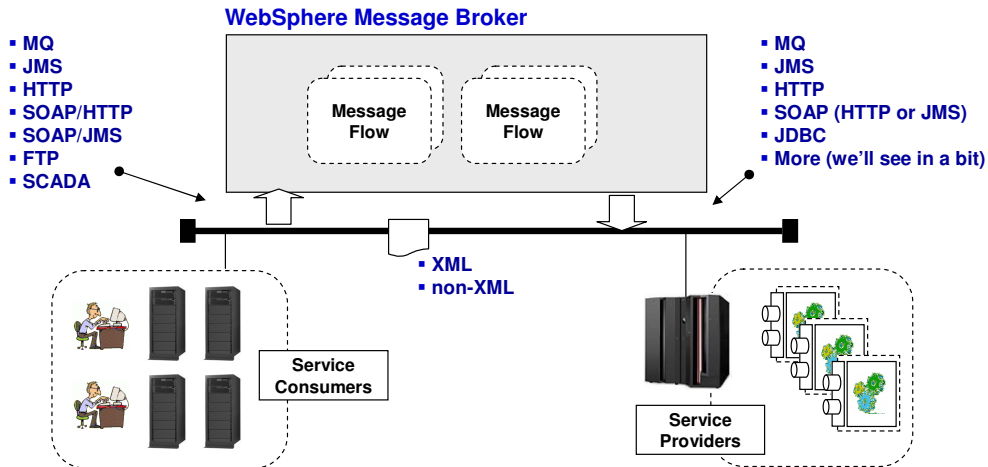
© 2007 IBM Corporation

If we start this discussion at a really high level, we see that WebSphere Message Broker (WMB) is a lot like WESB, at least conceptually. The text along the right side of the chart summarizes how, at the conceptual level, the two are similar.

The physical picture also illustrates this -- both products are middleware implementations of the ESB concept which maps onto your corporate network and provides an intermediary between your service consumers and your service providers.

## What WebSphere Message Broker Is

It is a powerful message flow runtime environment, with a wide range of connectivity options:



### Much more to explore:

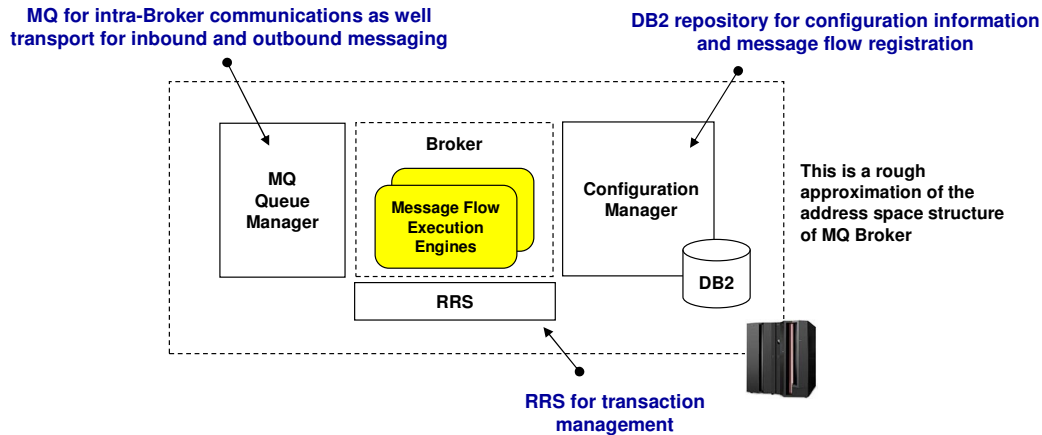
- How a message flow is constructed
- What comprises a flow
- How a flow is deployed and executed

The Physical Implementation ...

WebSphere Message Broker is a "message flow" runtime execution environment. We'll go into a great deal more detail about what a "message flow" is. WMB has a remarkable suite of connectivity options on the front end, and an impressive list of connectivity options on the back end. It takes as a message format XML as well as non-XML, and have a powerful capacity for message transformation within the Broker itself.

## The Physical Implementation of WMB on z/OS

**WMB is built on MQ. The physical structure looks like something like this:**



**What goes on *inside* the Broker is what's really interesting.  
The WMB message flows are incredible powerful.**

**Focus on programming capabilities ...**

35

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

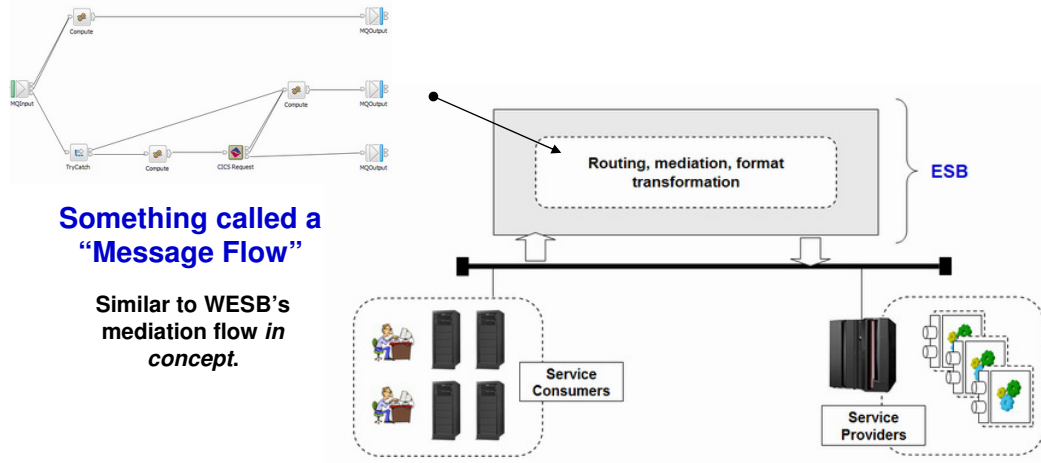
WebSphere Message Broker is built upon the MQ base. As such, it's comprised of a Queue Manager, a "Configuration Manager" (along with some backing database tables in which the configuration and message flow information is maintained) and the Broker itself. This picture is an approximation of the actual address spaces used for WMB -- in truth the "Broker" piece of this is multiple address spaces, depending on how many "execution groups" you have defined.

The Queue Manager is used for the handling of message-based communications, either from outside of WMB or within WMB. The Configuration Manager is the coordination point for managing the configuration of the system. The Execution Groups are where message flows actually run.

This picture puts WMB into a "real" perspective for you. But it doesn't really tell the story of what goes on inside of WebSphere Message Broker. That's the story we tell now.

## Message Handling Intelligence in the ESB

Again, we need to talk about the programming capabilities of the ESB



**WMB's built in capabilities are far more extensive than WESB's. Much of this story is going to be told by reviewing these built in capabilities.**

**Broker Toolkit ...**

Just like with WESB, we can't really tell the story of WMB without going into some depth on the programming capabilities of the product. That's where the power is going to become evident. For WMB, that means we're going to talk about "message flows," which are similar in concept to the "mediation flow" of WESB. WMB is not SCA, however, and it has more capabilities.

## Broker Toolkit

Yet another “Eclipse-based” tool used to create WMB message flows:

**“Palette” that contains the various nodes**

**Message Flow Editor -- graphical drag/drop environment**

**Property settings for the selected node**

**Nodes ...**

**Rational Application Developer**

**Rational Web Developer**

**WebSphere Developer for zSeries**

**WebSphere Integration Developer**

**WebSphere Message Broker Toolkit**

**Eclipse**

**This is like any graphical environment -- takes some time to get proficient.**

**You’ll see all of this in lab**

**The deployment artifact is the BAR file ... Broker ARchive**

**Nodes ...**

37

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

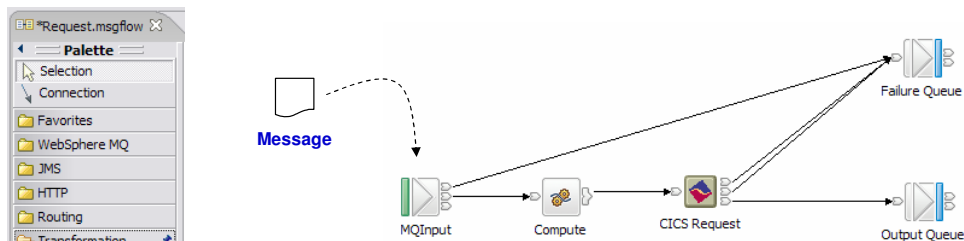
The tool used to create a WebSphere Message Broker “message flow” is the “Broker Toolkit.” It is yet another Eclipse-based tool. We refer back to our picture showing where the Broker Toolkit sits in relation to other things we’ve seen -- Rational Application Developer, WebSphere Developer for zSeries, WebSphere Integration Developer, and now WebSphere Broker Toolkit.

It is in the Broker Toolkit that you’ll “draw out” your flow and set the properties. Like any graphical environment it’ll take a little getting used to before you become comfortable. (Recall your first experience with Powerpoint and recall how intimidating it was initially.) You’ll get an opportunity in the lab to do just this -- create a message flow. It’ll seem like a lot of point-and-click, but our hope is you’ll at least see the basic concepts at work.

The deployment artifact is the BAR file. It is what gets deployed into the Message Broker. They run in the Execution Engines

## Nodes

**Nodes are the basic building blocks of a message flow. They represent functional routines that encapsulate the flow logic. Nodes are used to create a flow, which represents the “reusable integration application” inside the ESB:**



### Key Points:

- Each node has a set of properties, such as the MQ queue name, or the URL of the requested web service, etc.
- Compute nodes contain your integration logic  
     Compute = ESQL (WMB Language)  
     Java Compute = Java
- Many nodes exist that are not shown here  
     Nodes that come with WMB  
     Nodes you can add to WMB

Broker Toolkit offers a “Palette” with nodes available for use

Built-in Nodes ...

38

IBM Americas Advanced Technical Support  
 Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

We start the discussion by focusing on “nodes,” which are the elemental building blocks of a message flow. A node represents some kind of functional routine that’s used to act upon a message. Each node has an input terminal and some number of output terminals (depending on the node). Connections between nodes define how a message is to flow.

**Note:** Does this look familiar? It should ... the SCA model is very similar to this. But it should be noted that WMB is not SCA-compliant. WMB uses its own graphical-abstraction model. It’s similar to SCA, but it is not SCA.

You’ll “draw” these nodes onto the Broker Toolkit “canvas” by selecting the nodes from a “palette.” If you’ve done any work at all in something like Powerpoint, you’ll get the drag-and-drop nature of this very quickly.

The keys to this are as follows:

- Each node has a set of properties that you define to it. For instance, a MQInput node has properties that define the queue name, the message type expected, whether the Broker is to maintain it within a transactional scope. A good deal of the effort of creating a message flow is knowing what properties to set and how to set them.
- Compute nodes are what contain your integration or transformation logic. There are two basic compute nodes -- a Java compute node which takes Java as its programming language; and the regular Broker compute node which takes Broker’s programming language: ESQL. We’ll talk more about ESQL, but for now understand it’s a kind of procedural language.
- This picture is showing only a small handful of nodes. We’re about to show you a whole bunch more. There are two basic types of nodes -- those that are “built-in” to the Toolkit; that is supplied with the tool ... and those that can be downloaded and installed into the Toolkit to provide additional functionality.

**Built-in Nodes: Input/Output Related**

The Broker Toolkit comes with a set of built-in nodes you can use to start building message flows right away. Additional nodes are downloadable. The downloadable nodes come in the form of SupportPacs. More in a bit.

MQ	HTTP	JMS	Telemetry and Real Time
Read message off queue MQInput	Receive HTTP request HTTP Input	Receive from JMS destination JMSInput	Receive from telemetry device SCADAInput
Write message to queue MQOutput	Send web service request to URL HTTP Request	Reply to JMS destination JMSOutput	Receive to telemetry device SCADAOutput
Read message off queue MQGet	Reply to original requester HTTP Reply	JMS to MQ JMS transform JMSMQTransform	Receive over MQ JMS real time transport Real-timeInput
Send response to queue indicated by requester MQReply		MQ JMS to JMS transform MQJMSTransform	Real time pub/sub Real-timeOptimizedFlow
Related to pub/sub MQOptimizedFlow			

The WMB InfoCenter has an excellent description of what each of these nodes do. Just search on node name. URL for InfoCenter:  
<http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp>

**More built-in nodes ...**

39 IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD © 2007 IBM Corporation

We're now going to on a survey of the various nodes that are part of Broker's functionality. And we're going to do this first by built-in nodes, then by add-on nodes that are supplied as something called a "SupportPac." There are too many built-in nodes to include on one chart, so we'll do a survey by type. We start with Input/Output related nodes.

This we break down into four categories: MQ, HTTP, JMS and Telemetry/Real-Time. Space in Powerpoint's "speaker notes" function won't allow a full description of each.

- **MQ** -- The basic ones are MQInput and MQOutput, but others exist as you see on the chart. If the flow originates and ends with a message to/from a queue, you'll make use of these. Within the properties of the node you set the queue name.
- **HTTP** -- three types here: HTTP Input (which would be used when the request flows in over HTTP); HTTP Request (when you want to invoke an external HTTP site to get information ... it can be a Web Service but it doesn't have to be); and HTTP Reply when you want to reply back to the original requester ... much like a normal web site does.
- **JMS** -- JMSInput and JMSOutput are used when JMS is the transport mechanism; the two transform nodes modify the header so a message can flow between JMS and MQ.
- **Telemetry and Real Time** -- nodes we might not normally think about, but they're heavily used in the process and monitoring industry. SCADA is an industry standard protocol for picking up input from telemetry sensors. The MQ "real time" support maps to MQ's function of the same name.

The WMB InfoCenter is an excellent source of information on these things. The URL on the chart gets you to the front page, and from there you can do a search.

## Build-in Nodes: Routing, Transformation and Database

Routing	Transformation	Database
<b>Filter</b> Route based on message content	<b>Compute</b> Construct new output message with ESQL	<b>Database</b> Interact with database using ESQL
<b>Label</b> Used with RouteToLabel	<b>JavaCompute</b> Construct new output message with Java	<b>DataInsert</b> Insert data into table
<b>Publication</b> Route to topic subscribers (pub/sub)	<b>XMLTransformation</b> Transform XML using XSL	<b>DataDelete</b> Delete data from table
<b>RouteToLabel</b> Route to label node	<b>Mapping</b> Populate message with new content	<b>DataUpdate</b> Update data in table
<b>AggregateControl</b> Beginning of message fan-out	<b>ResetContentDescriptor</b> Forces reparsing of the message	<b>Warehouse</b> Store entire message (or part) in database
<b>AggregateReply</b> End of a fan-in		
<b>AggregateRequest</b> Related to fan-in/fan-out messages		

**Those were the “built-in” nodes. More nodes are possible. They can be added to your Toolkit. They come packaged as “SupportPacs”**

**SupportPacs ...**

40

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

We explore more:

- **Routing** -- these are used to affect the routing of messages within a flow. The most common one is the Filter node, which is similar in concept to the MessageFilter primitive from WESB. This will route based on message content. The other nodes are for things like “publish/subscribe” and routing to multiple endpoints.
- **Transformation** -- these are the message handling workhorses. These you can “program” with code to do custom modification of the messages as they go through the message flow. For non-Java the most common would be the “Compute” node, which takes as input Broker’s ESQL language. (We have more to say about ESQL in a bit.) For Java, the JavaCompute node is provided. Like WESB, Broker has an XMLTransformation node that uses XSLT. The Mapping node can be used to indicate how message content can be remapped to different content.
- **Database** -- nodes that provide JDBC access to a relational database. This can be used to log information, or extract information you want to put into a message, or to validate some portion of a message based on information provided in a database table.

Those are the “built-in” nodes. Now we’ll look at some of the “SupportPac” nodes.



**WebSphere MQ SupportPacs**

**Downloadable code and documentation that complements the WebSphere MQ family of products, including Message Broker.**

<http://www.ibm.com/software/integration/support/supportpacs>

**Business Integration - WebSphere MQ SupportPacs**

**Product documentation**

**Abstract**  
IBM® WebSphere® MQ family SupportPacs™ provide you with a wide range of downloadable code and documentation that complements the WebSphere MQ family of products. The majority of SupportPacs are available at no charge to users. Others can be purchased as fee based services from IBM.

**Content**  
[Overview of the WebSphere MQ SupportPac system](#)

Select SupportPacs by:

**Category:**  
[Category 1 - Fee based services](#)  
[Category 2 - Freeware](#) ← **The utility we'll use in lab to put a message on an MQ queue comes from here (IH03 SupportPac)**  
[Category 2 - Performance Reports](#)  
[Category 3 - Product extensions](#) ← **The CICS Request node we'll use in lab (IA12) comes from here**  
[Category 4 - Third party contributions](#)

**Product:**  
[WebSphere MQ](#)  
[WebSphere MQ Everyplace](#)  
[WebSphere Data Interchange](#)  
[WebSphere BI Adapters and Collaborations](#)  
[WebSphere Message Broker](#)  
[WebSphere MQ Workflow](#)  
[WebSphere Partner Gateway](#)

**Others:**

- FTP
- POP3 e-mail
- WSRR
- TCP sockets
- etc.

**Provided in the form of a ZIP file.**  
**Installation differs from node to node:**  
 Use "Install Features" of WMB  
 Unzip directly into file structure  
 (Each comes with PDF instructions)

**z/OS specific nodes ...**

41 IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD © 2007 IBM Corporation

WebSphere MQ SupportPacs is a way to supply additional functional to customers outside the maintenance stream. The URL at the top of the chart provides the location where you can go for the main MQ SupportPac page.











There are four "categories" of SupportPacs. The ones we're interested in for this workshop are Categories 2 and 3 -- Freeware and Product Extensions. In lab we're going to use a tool called "rfhutilc," which is really a workstation graphical tool that's really handy for placing messages onto an MQ queue and reading them back off. That's a Category 2 SupportPac called IH03. In lab we're also going to use the CICS Request node, which is packaged as the IA12 SupportPac and is supplied as a "Product Extension" Category 3. There are more Category 3 packs that provide nodes to Broker -- FTP, POP3, WSRR (WebSphere Services Registry and Repository ... we'll talk about that more in a little bit), and TCP sockets.

You should be able to see the basics of this -- IBM supplies these things as optional downloadable function you install into your Toolkit for use if you desire that function. Each SupportPac installs in a slightly different way, so it's always best to read the README and the PDF that comes with each.

Now let's look at the z/OS-specific nodes that are available.

## z/OS Specific Nodes

**These nodes are designed to interact with specific z/OS resources.**

<p><b>IA13</b> <b>VSAM</b></p>  VSAMInput  VSAMRead  VSAMWrite  VSAMUpdate  VSAMDelete <p><b>Has a z/OS-side component that needs to be installed</b></p>	<p><b>IA11</b> <b>File Adapter for z/OS sequential files</b></p>  FileDelete  FileRead  FileRename  FileWrite <p><b>File format:</b>        •QSAM        •F,FB,V or VB only  <b>Has a z/OS-side component that needs to be installed</b></p>	<p><b>IA12</b> <b>CICS</b></p>  CICS Request <p><b>Provides local access to CICS</b>        (Note: there are other ways to access CICS without this node. More on that later.)</p>
---	--	--

**These SupportPacs extend the function of WMB to do z/OS-specific things**

**Parsing the received message ...**

42

IBM Americas Advanced Technical Support  
 Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

Here we take a look at a few SupportPacs related specifically to z/OS.

- **IA13** -- VSAM nodes. This is used to access VSAM datasets on z/OS. You can read or update, add or delete information from a VSAM dataset. This support pack requires a z/OS-side component be installed. That's what gives the SupportPac the necessary API support on z/OS to do what it's trying to do.
- **IA11** -- Sequential datasets ... specifically, QSAM data sets with F, FB, V or VB record format only. Here again, a z/OS-side component needs to be installed.
- **IA12** -- CICS Request. This SupportPac we'll use in class. With it, you can invoke a CICS transaction using the EXCI interface of CICS. The results come back in COMMAREA format, which your flow will then need to handle.

There's a very important element of this story we need to cover next. It has to do with how Broker reads in (or "parses") a received message so it can create the "Logical Message Model" (a concept very similar to what's used by WESB and its "Service Method Object".)

## Parsing the Message ... and the Logical Message Model

The Input node of a message flow parses the message and constructs a “logical message model” -- a structured representation of the data:

```

01 TRANSACTION.
  10 CUSTOMER.
    15 FIRST-NAME PIC X(15) .
    15 LAST-NAME  PIC X(30) .
  10 ADDRESS.
    15 STREET      PIC X(30) .
    15 CITY        PIC X(25) .
    15 STATE       PIC X(20) .
    15 ZIP         PIC X(5) .
  
```

Similar in concept to the Service Data Object we saw for WESB

- **Input node has built in parsers:**
  - BLOB – No structure, just a sequence of bits
  - XML, XMLNS, XMLNSC – Self defining, generic XML message
  - MRM – Fixed record structures, tagged/delimited or XML
  - JMS – Standard folder structure for a JMS message
  - These parsers can be supplemented with user written custom parsers or those purchased from a third party
- You can define modeling by importing COPYBOOK, C Header Files or XML DTDs or schema
- This provides a consistent model throughout WMB, and provides for very fast data access.

**A lot of flexibility here ... WMB06 workshop goes into far more detail.**

ESQL ...

43 IBM Americas Advanced Technical Support Washington Systems Center, Gaithersburg, MD © 2007 IBM Corporation

Message Broker cannot assume that every message is going to be received with its data formatted in exactly the same way. That should be obvious. So it's necessary for Broker to read in -- to “parse” -- the incoming message. This is something done by the input node, and it's related to a broker concept called the “Message Set”. You'll get to work with “Message Sets” in the lab, but in summary it is a way for Broker to be told what to expect of the message coming and, and how to parse it.

Why parse it at all? Because what we're trying to get at is a structured representation of the data in the message so subsequent nodes can access the data in a much easier fashion. If the text was just a string of characters, we'd have to tell subsequent nodes how far to go into the string and how many characters a given piece of data actually was. But if we construct up a “logical message model,” this allows nodes to quickly address data in the message structure. In the picture above the customer's last name is accessible via:

```
Root.Transaction.Customer.LastName
```

**Note:** that's *not* an exact representation of the syntax of accessing data, but it's close. In lab, if you look at the ESQL being used, you'll see syntax somewhat similar to that.

How can we tell Broker what the incoming message is going to look like? It depends on where the message is going to come from. If the message is coming in the form of a return from CICS, we can supply the Broker Toolkit with the COBOL COPYBOOK which provides the language structure of the message. The Broker Toolkit can then generate code which is used by the message flow to parse the received message and construct the logical message model accordingly.

There's a lot more to this. The concept of the “logical message model” is very key to the Broker's functionality. Accessing data in the message, modifying data in the message, using data from the message to do lookups -- all rely on a logical message model. As mentioned, the concept is very similar to Service Data Objects and Service Method Objects from Services Component Architecture.

## ESQL - Extended Structured Query Language

ESQL is a programming language defined by WebSphere Message Broker to define and manipulate data within a message flow.

Can use ESQL in any of the following nodes:

- Compute node
- Database node
- Filter node
- DataDelete node
- DataInsert node
- DataUpdate node
- Extract node
- Mapping node
- Warehouse node

**Compute Node**

```

IF (XML format required) THEN
  OutputRoot.Properties.MessageFormat = 'XML';
ELSE IF (custom format)
  OutputRoot.Properties.MessageFormat = 'CWF';
ELSE IF (SWIFT format)
  OutputRoot.Properties.MessageFormat = 'TDS';
ENDIF;

```

**Data Insert Node**

```

IF Body.Person.height > 183 THEN
  INSERT INTO Database.TallPeople
    (Name, Height, Age)
  VALUES (Body.Person.Name,
    Body.Person.height,
    Body.Person.age);
ENDIF;

```

**Data types**

INTEGER  
FLOAT  
DECIMAL  
STRING  
DATETIME  
BOOLEAN  
REFERENCE  
NULL  
...

**Operators**

- + \* /  
||  
AND OR NOT  
= < > >= < <=  
IN BETWEEN  
LIKE  
IS EXISTS

**Statements**

**Basic**  
DECLARE  
SET  
IF ENDIF  
WHILE  
  
**Tree**  
MOVE  
CREATE  
DETACH  
ATTACH  
  
**Database**  
INSERT  
DELETE  
UPDATE  
PASSTHRU  
EVAL  
  
**Node**  
PROPAGATE  
RETURN  
THROW  
...

**Functions**

**String**  
LENGTH  
TRIM LTRIM RTRIM  
OVERLAY  
POSITION  
SUBSTRING  
UCASE LCASE  
  
**Numeric**  
ABS  
BITAND NOT (X)OR  
MOD ROUND  
SQRT  
TRUNCATE  
EXTRACT  
  
**Datetime**  
EXTRACT  
CURRENTDATE  
CURRENTTIME  
  
**Field**  
BITSTREAM  
CARDINALITY  
FIELDTYPE  
SAMEFIELD  
  
**Complex**  
CAST  
SELECT  
...

**Remember - Java Compute Node available to leverage Java programming skills**

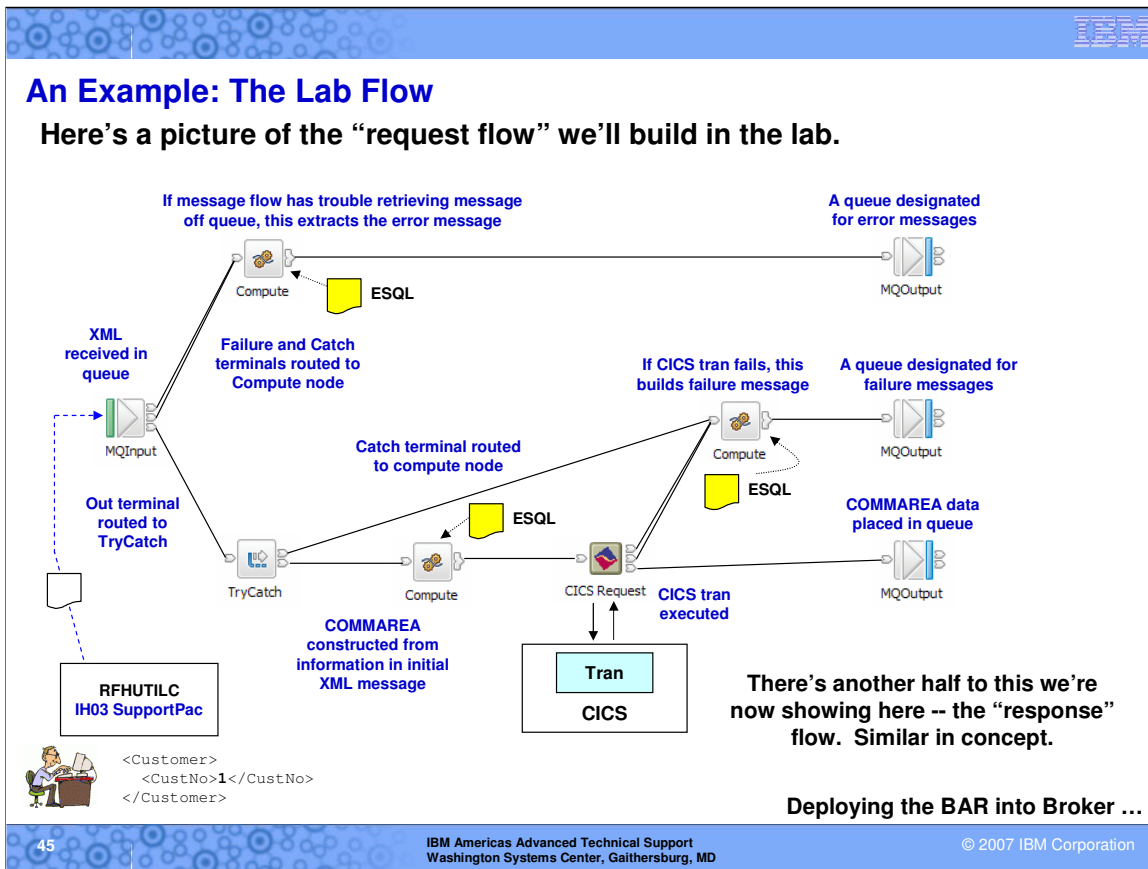
**The flow we'll build in lab ...**

44
IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD
© 2007 IBM Corporation

ESQL is Message Broker's structure query language for its nodes, including the Compute node and the nodes shown at the upper left of the chart. ESQL is a kind of procedural language, tailored to be particularly powerful at handling access to data contained within a structured, logical message model. (Hence the focus on the Logical Message Model on the previous page.) With it you can query data elements within a message, change them, do computation work on them, substring out smaller increments ... pretty much anything you can do with any programming language.

**Note:** The Java compute node is provided for those who wish to leverage Java expertise.

To illustrate how and why ESQL would be used, let's look at the flow we're going to build in lab. We'll show where ESQL we're going to supply you will be employed in the flow.



This is actually one-half the flow we'll build in lab. This is the request flow -- the flow that will be executed when the request is first received from the service consumer. We see that ESQL is used in three different compute nodes in this flow. Let's look at what's going on:

- The input message is going to consist of a simple XML with an integer representing a customer number. We're going to use the IH03 SupportPac to put that message on the input queue.
- Going to the top of the chart -- if for some reason Broker has trouble pulling the message off the queue, it's going to go up to the compute node where ESQL will extract the error message, then route the error message to a queue designated for that purpose.
- Assuming Broker pulls the message off the queue okay, the message will flow down to the compute node just prior to the CICS Request node. There the input -- XML -- will be formatted into the appropriate COMMAREA format so the CICS request can be done. The reformed input message then flows to the CICS Request node. If the processing in the compute node fails, the failure flows back to the "TryCatch" node and then up to the other compute node where the failure message will be extracted and flowed over to the failure queue.
- The CICS Request node then issues the request against CICS. If things work okay, the returned COMMAREA message is then placed on the queue for processing by the response flow, which is *not* shown on this chart (not shown because things would get too busy on the chart). But if things fail in the request to CICS, then the failure flows up to the compute node where the failure message (this is a different failure message from the one possibly thrown by the earlier compute node) is extracted and flowed over to the failure queue.

So you see three different uses of ESQL -- (1) to extract the error message, (2) to transform the input XML to COMMAREA, and (3) to extract the failure message.

## Deploying the Packaged Message Flow

This is done from the Broker Toolkit, which has a connection to the Configuration Manager region of Broker up on z/OS

- The BAR file is packaged by the Toolkit
- The Domain Connection defines the host and port of the CMGR
- The BAR file is transferred to the CMGR
- The flow is deployed into the Broker
- The CMGR updates the DB2 repository with information about the deployed flow

**The Broker can handle many deployed message flows, just like WESB could.**

Relating the request to the flow is similar to how WESB does it:  
 URL defined to HTTP input node  
 Queue on which a message is received

**The Big Picture ...**

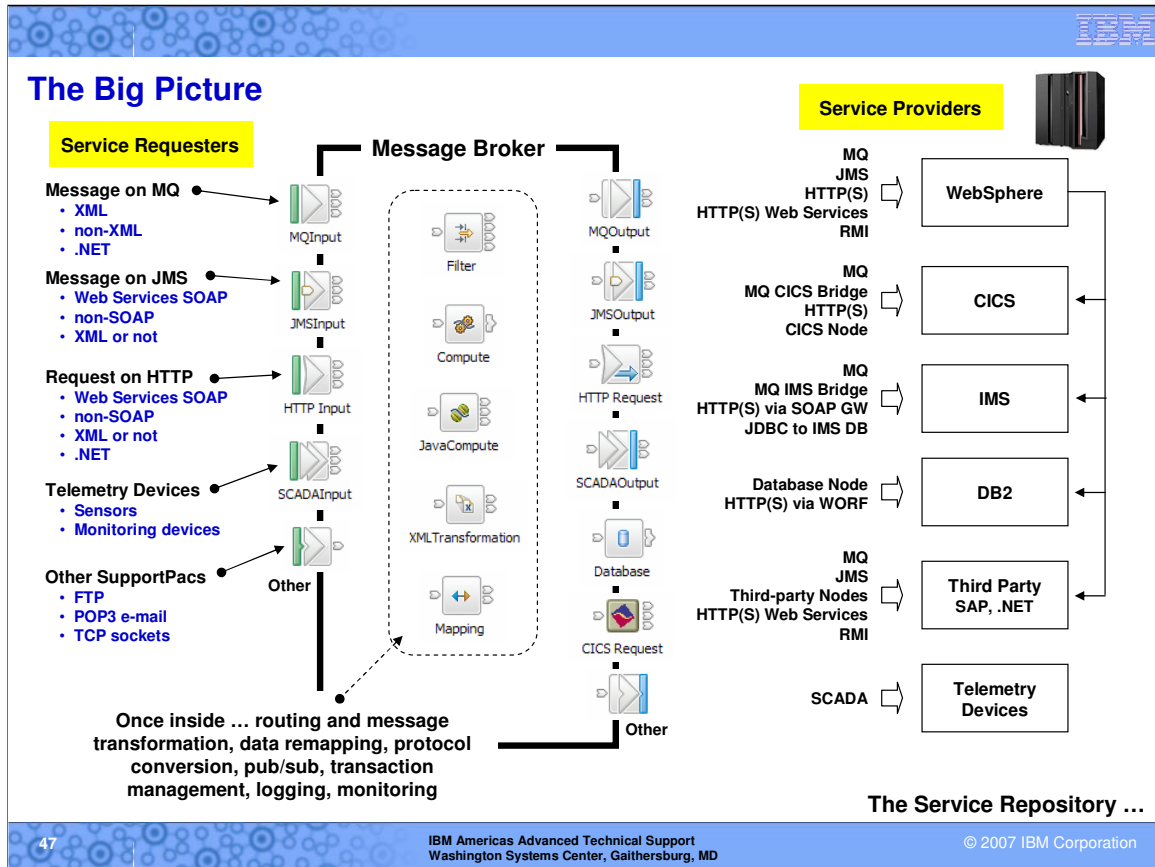
46 IBM Americas Advanced Technical Support  
 Washington Systems Center, Gaithersburg, MD © 2007 IBM Corporation

Deploying the constructed message flow -- held in a BAR file -- is done from the Broker Toolkit. One of the things the Toolkit provides is "Domain Connection" to the Broker's configuration manager running on z/OS, with the ability to "Deploy Archive" to a selected execution group. An execution group relates to an address space on z/OS where the flow will run.

The act of deploying the BAR file really involves two things: deploying the artifacts into the Broker's execution engine; and updating the configuration manager so it's aware of the deployment and the current status of the flow.

The Broker is capable of hosting many deployed message flows ... not just one. And many can operate within any given execution engine. So how does WMB differentiate one flow from another when a message is received? In a manner very similar to how WESB did. It's all based on the definition of the input node. If an MQInput node, then receipt of a message on the queue referenced in the properties of the MQInput node will trigger the execution of that flow. WMB maintains an awareness of what flows are associated with what queues. Similarly, a flow with an HTTP input node will associate the URI value just like WESB did.

Let's now look at the big picture ...





This picture is trying to illustrate all the different ways to flow input to the Broker, what can be done inside the flow, and what connections are available to backend data resources. This is a complex picture, but it shows the power of WebSphere Message Broker.

- On the input side, several different nodes provide input via HTTP, MQ, JMS or other forms. The input can be standardized Web Services, non-standard format, or even .NET. The message format itself can be XML or not.
- On the inside, there are nodes to affect the routing a message flow takes; nodes to transform the message content, message format or protocol; there are nodes to insert data, subtract data or remap data. And remember, while traversing the flow, the message is in the "logical message" format, which means access is structured and fast.
- On the service provider side nodes are present to give access via MQ, JMS, HTTP, JDBC or direct access to CICS. Coming out with MQ, JMS, HTTP -- Web Services or not -- we have access to WebSphere, CICS, IMS, DB2, third party products and telemetry devices. And overarching the whole thing is the knowledge that if the access to WebSphere initially, then WebSphere itself has all sorts of connector support to get to CICS, IMS, DB2 or third party products.

There you have the "big picture" -- a powerful message transformation and routing function, implemented in middleware, that rides on top of your existing networking infrastructure. Earlier we defined ESB as just that. Here we see WebSphere Message Broker as an ESB ... many call it the "Advanced ESB" because of its capabilities.

We're now ready to investigate a "services repository" function -- WSRR.





**Small Detour**

# WebSphere Services Registry and Repository (WSRR)

48

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

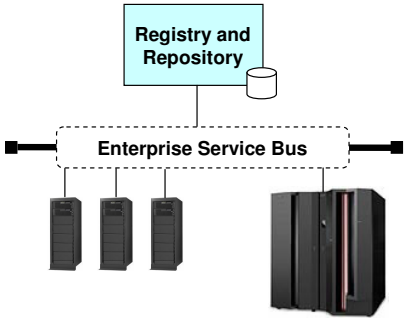
© 2007 IBM Corporation



## Setting the Stage for Discussion of “Registry and Repository”

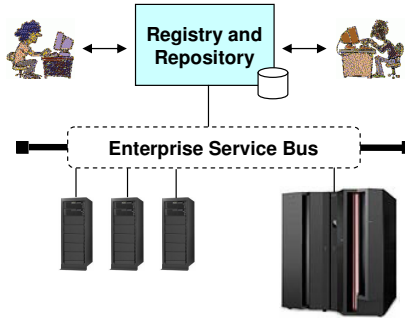
There are two fundamental things a registry and repository is going to provide:

As a runtime source of information,  
*retrieved programmatically*



This is perhaps the more common use of a registry people think about ...

As a **organized** and **centralized** source of information for planning, developing and managing the SOA environment



... but overlooking this means you'll miss what IBM is planning for this function.

**Imagine trying to control this if the information was scattered across different notebooks, spreadsheets, yellow sticky notes on the wall, in their heads, etc.**

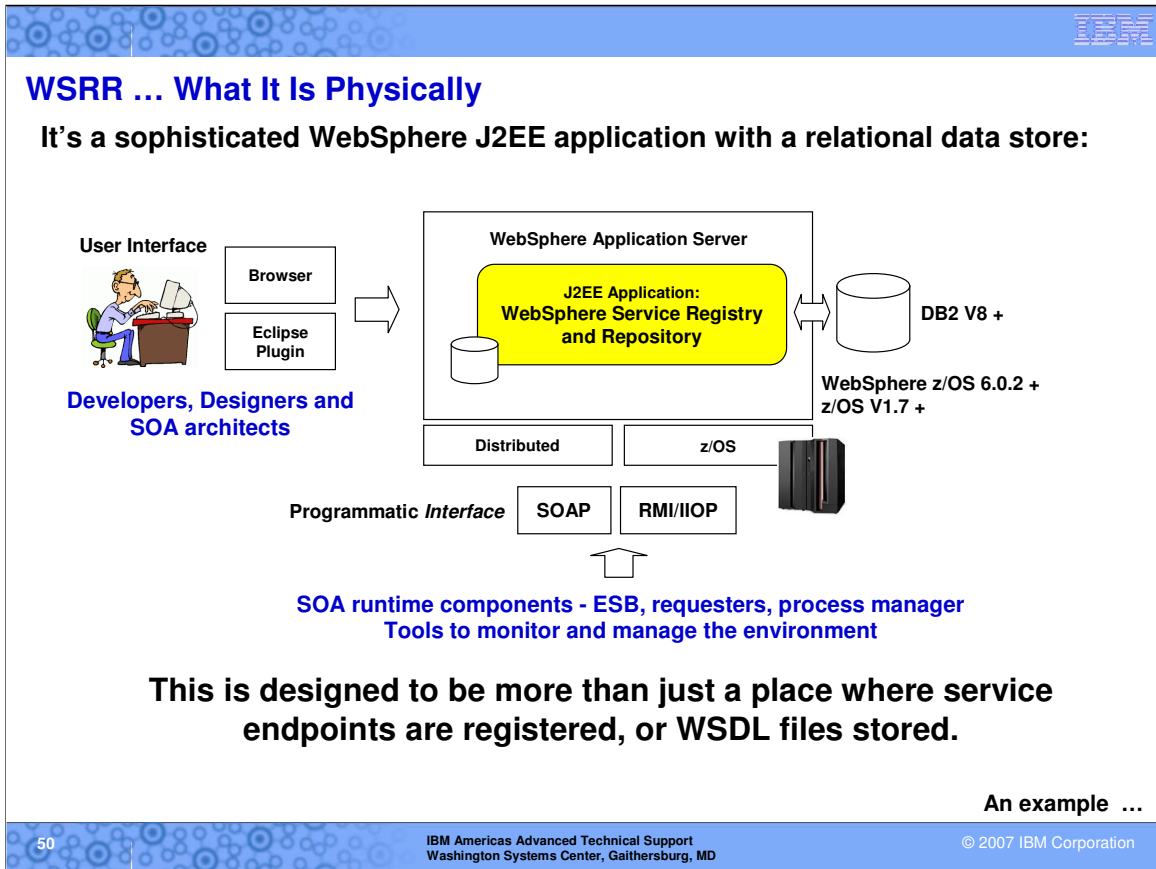
**WebSphere Service Registry and Repository ...**

49
IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD
© 2007 IBM Corporation

To begin the discussion of the WebSphere Service Registry and Repository function, it's useful to consider the two fundamental things a registry and repository provides: one is a place where runtime components can go and retrieve information programmatically so things are more dynamic in that environment. This is probably the first thought that comes into people's minds when they think "registry" and "SOA."

But the second thing the registry and repository function does is provide a organized and centralized place for information about the SOA. This is really quite critical because without it an SOA beyond a small pilot implementation would quickly become unmanageable. Imagine an environment where this kind of information was scattered across a dozen different places, some accessible and some not. That's the world you'd be facing without a repository.

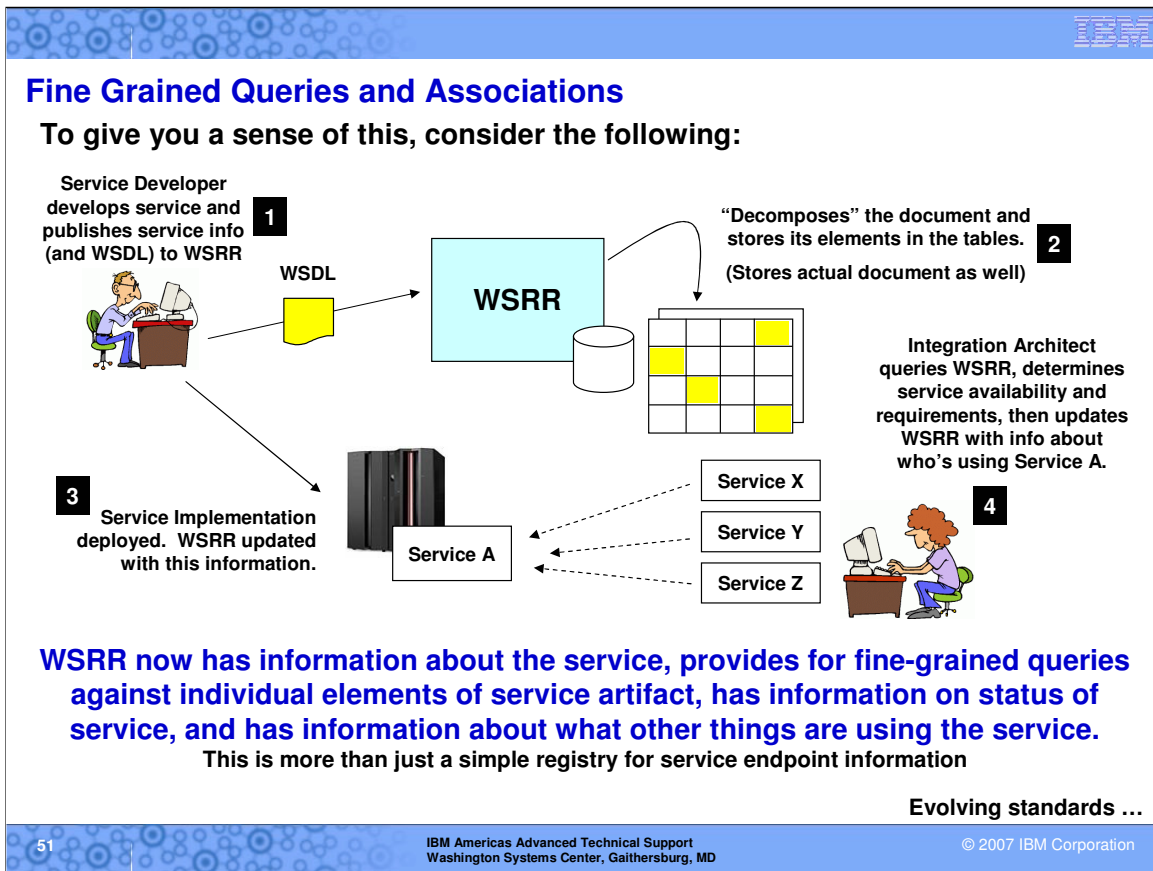
This second reason for a registry and repository is what IBM has in mind for the WebSphere Service Registry and Repository (WSRR) product. It's not *just* a service endpoint registry (though it is that ... it not *just* that), but also a place where the information can be gathered and structured so it can be used for planning and control. This maps directly to the governance topic we'll discuss later and the four stages IBM proposes: model, assemble, deploy, manage. Each one of those phases involves using and updating information. A single, well-organized place for that information is critical.



Our first picture showing WSRR will be a physical representation. We do this because we want very much for you to overcome the somewhat mysterious nature of this thing and come to understand the essentials of it.

WSRR is, at its core, a sophisticated EJB application that runs inside of WebSphere Application Server. It has behind it a sophisticated database definition. It provides human access in the form a browser interface and an Eclipse tool plugin interface. It provides programmatic access using either SOAP or RMI/IIOP. All this is designed to provide not just the runtime registry, but also the informational repository so things can be planned, modeled, monitored and queried in a controlled way.

Next we'll see a simplified example of how this might be used.

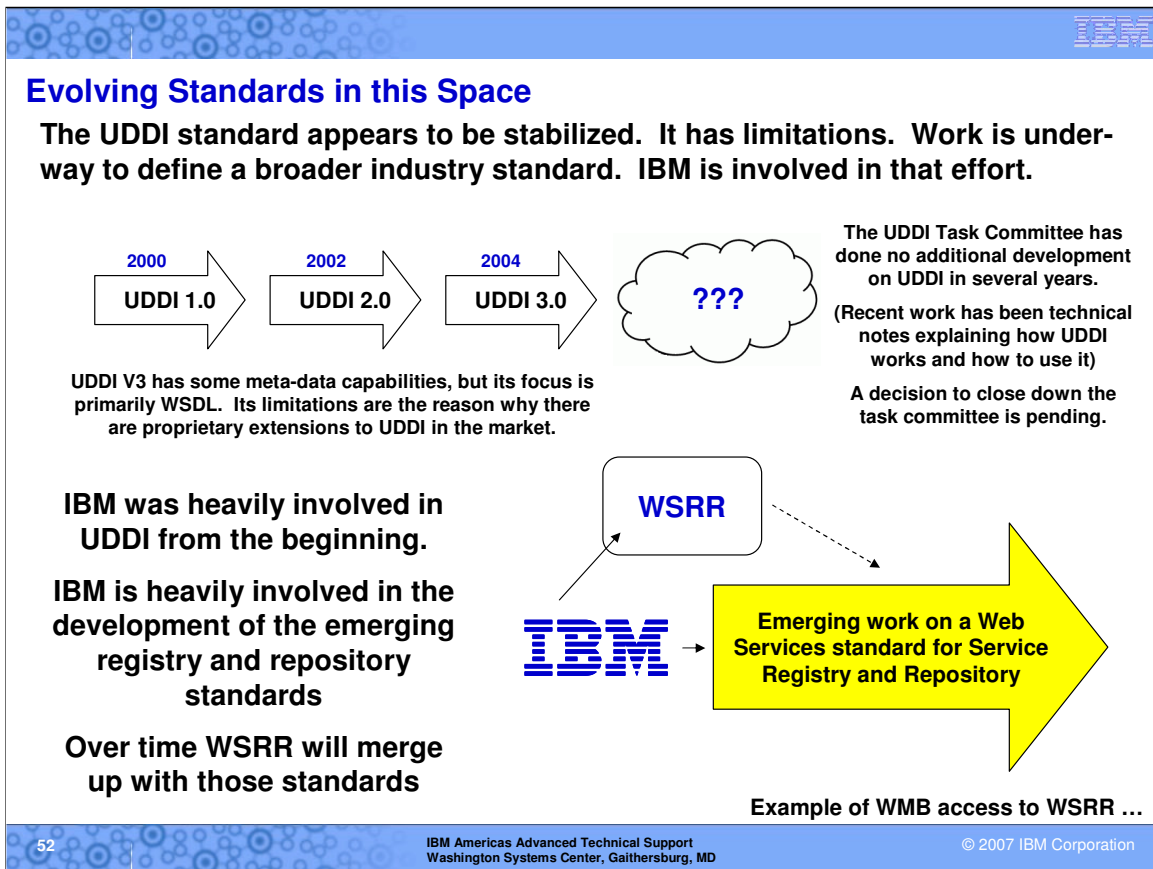


Here's an example -- somewhat simplified and made abstract -- that illustrates some of the key points about WSRR.

1. A developer creates a service implementation. Part of that is a WSDL file. The WSDL file is published to WSRR.
2. WSRR stores the WSDL, but it also decomposes the WSDL and stores its constituent pieces as well. This provides for relational queries on the elements, not just text searches through XML files.
3. The service is deployed to its runtime environment. The person who does the deployment also knows to update WSRR and provide the current status of the service: "Active" or whatever terminology the designers have allowed for the states of the services.
4. An integration architect queries the WSRR to see what services are out there and what capabilities they possess. This query can be against other "meta-data" that includes descriptions of the service, but also queries against the service definition elements. The integration architect determines that three other services could benefit from information provided by this new service "Service A." So the integration architect sets about updating the services X, Y and Z so they use Service A as part of what they do behind the scenes. The architect updates WSRR and provides it with information saying, in effect, "Service A is now used by Services X, Y, and Z."

So at this point the WSRR knows about the new service. It has all the information about that service included in its registry and repository, including a decomposed WSDL that can be queried at the element level. It has information about the status of the server ("Active"). And it has information about users of the service.

Can you see that what's happening here is there's a "picture" of the environment being maintained within a single, controlled and structured data store. Now imagine other tools being able to query this and provide reports, graphical representations, statistics, etc.



The inevitable question comes up -- "Why not just use UDDI?" The answer is: it could be ... up to a point. There are aspects of UDDI that are, apparently, a little short on flexibility to encompass all the demands a full SOA may place upon it.

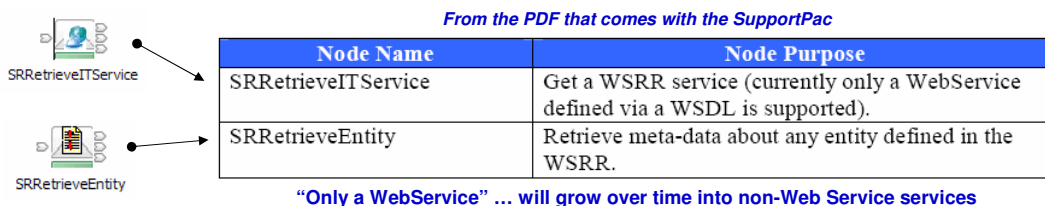
The history of UDDI is worth exploring. It came about back in 2000 and IBM was heavily involved in its formation. The latest version, 3.0.2, was finalized back in 2004 and since then no new development of the standard has taken place. What has happened is that some vendors have constructed "extensions" to UDDI to overcome some of its limitations. But as of the writing of this text, the UDDI task committee is not doing further development and there's a decision pending to close down the task committee.

That said, there is lots of activity around forming a new standard for a Service Registry and Repository that seeks to address the limitations of UDDI and provide a platform for full SOA. IBM is heavily involved in that as well. But the finalization of that is at least a year or more off. Rather than wait, IBM has provided WSRR as a solution in this space.

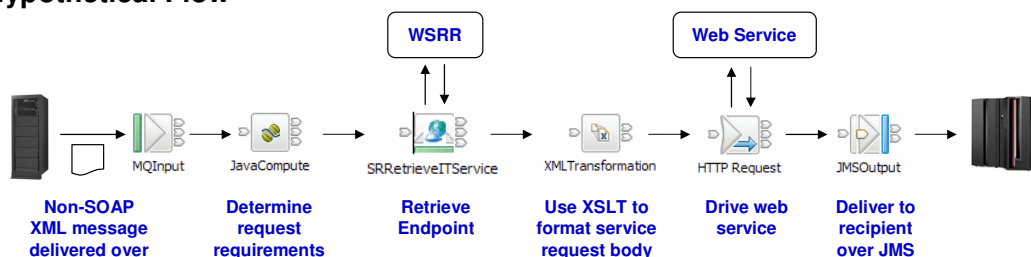
The key here is to understand that as the new standard evolves and shapes up, WSRR will evolve with it and take on more and more of the characteristics of the new standard. As we stated before, IBM is a strong champion of open standards. There's no reason to believe IBM won't be for this new standard.

## Example: WMB Access to WSRR

Two nodes are supplied with SupportPac IA9Q. One retrieves a WSDL file, the other retrieves “meta-data” about any service defined in WSRR:



## Hypothetical Flow



Back to the “Big Picture” ...

53

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

To give you an example of how a product can access and use the WSRR, there is a SupportPac available for WebSphere Message Broker. It's called IA9Q, and it's available at the same WebSphere MQ SupportPac website we mentioned a few charts back. Two nodes are supplied at the present time, one retrieves a WSDL file, the other retrieves “meta data.”

Notice how the support is limited to “Web Services only.” WSRR is a relatively new product, and other product support for this is an evolutionary thing. Web Services are likely to be the first kind of service people programmatically make available, so there's some justification in having the first support that comes out be for Web Services. One can expect that over time the support should expand to include other services.

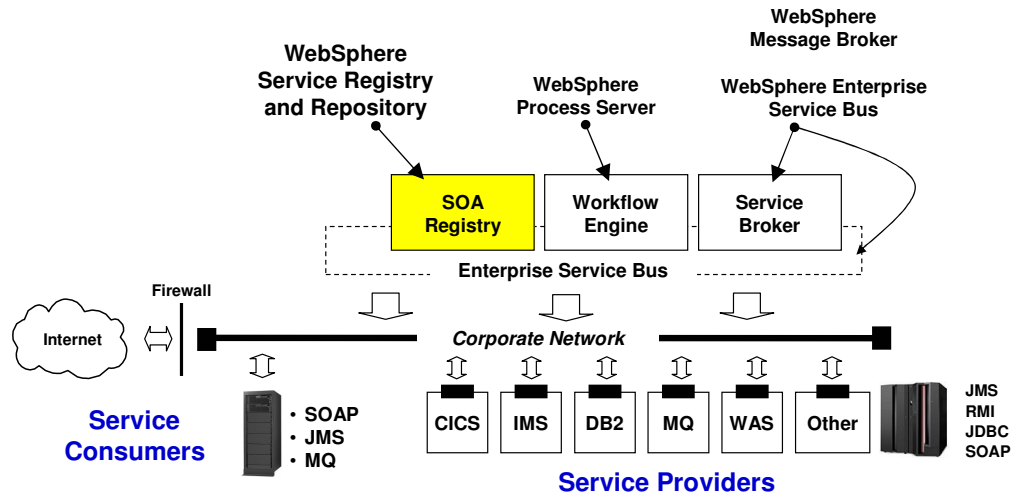
So let's look at a hypothetical Message Broker flow using this SupportPac:

**Note:** “hypothetical” because I've not actually built and run this flow. Essential concepts are correct; detailed specifics may not be entirely present.

- The flow starts with an MQInput node, where the received message is non-SOAP XML.
- The Java Compute node examines the XML to determine what the request is seeking. One element of the request is to be satisfied by driving a Web Service to get information prior to driving the final service provider.
- The IA9Q “SRRetrieveITService” node is used to query WSRR and get the endpoint information.
- An XMLTransformation node is used to transform the original XML to what the service requires
- An HTTP Request node is used to drive the Web Service and get the information.
- The message is then delivered via JMS to the final service provider.

## Where WSRR Fits in the Bigger Picture

WSRR implements the “SOA Registry” function ... one of the functions that expands the broader concept of an Enterprise Service Bus



Is WSRR a required piece of SOA or ESB? Strictly speaking, no. But it provides a very important function of SOA/ESB. You could write your own. Or use this.

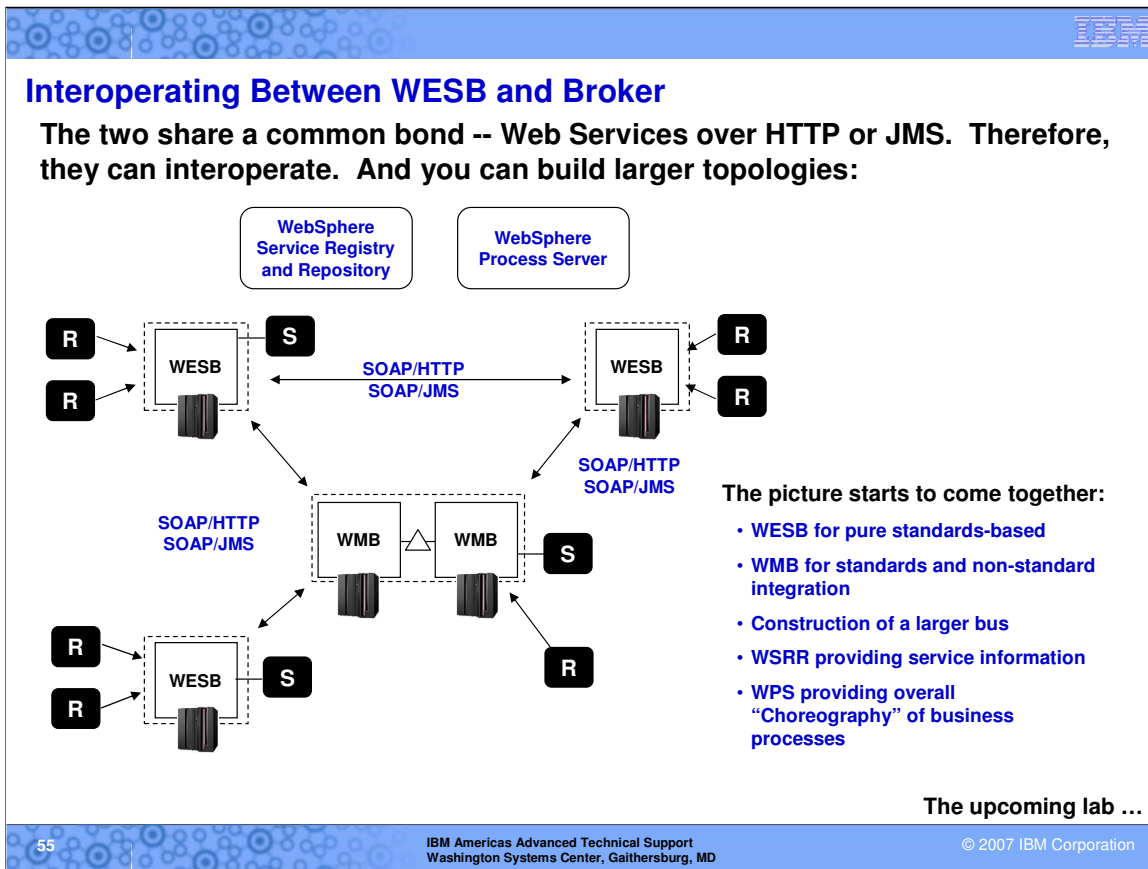
WESB and WMB interoperability ...

54

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

Here's our original picture showing a physical/logical representation of an SOA. The WSRR product fits above the ESB and is one of the key functions that extends the ESB. WSRR is not a required piece of the puzzle ... you can have an SOA or an ESB without using WSRR. But eventually you're going to need *some* kind of registry and repository. You could write your own, but in the end you'd have something that looks and operates in a way similar to WSRR. So using a pre-written product like this might save considerable time.



We've already shown that WESB and WMB both fill the role of "an ESB," and we've seen that WMB is often referred to as the "Advanced ESB" because of its capabilities beyond the standard Web Services arena. The two products do have an overlap of capabilities, and that common bond -- Web Services over HTTP or JMS -- can be used to interoperate between the two. So what we see is we can build out the ESB to span geographic locations or perhaps link different organizational divisions together.

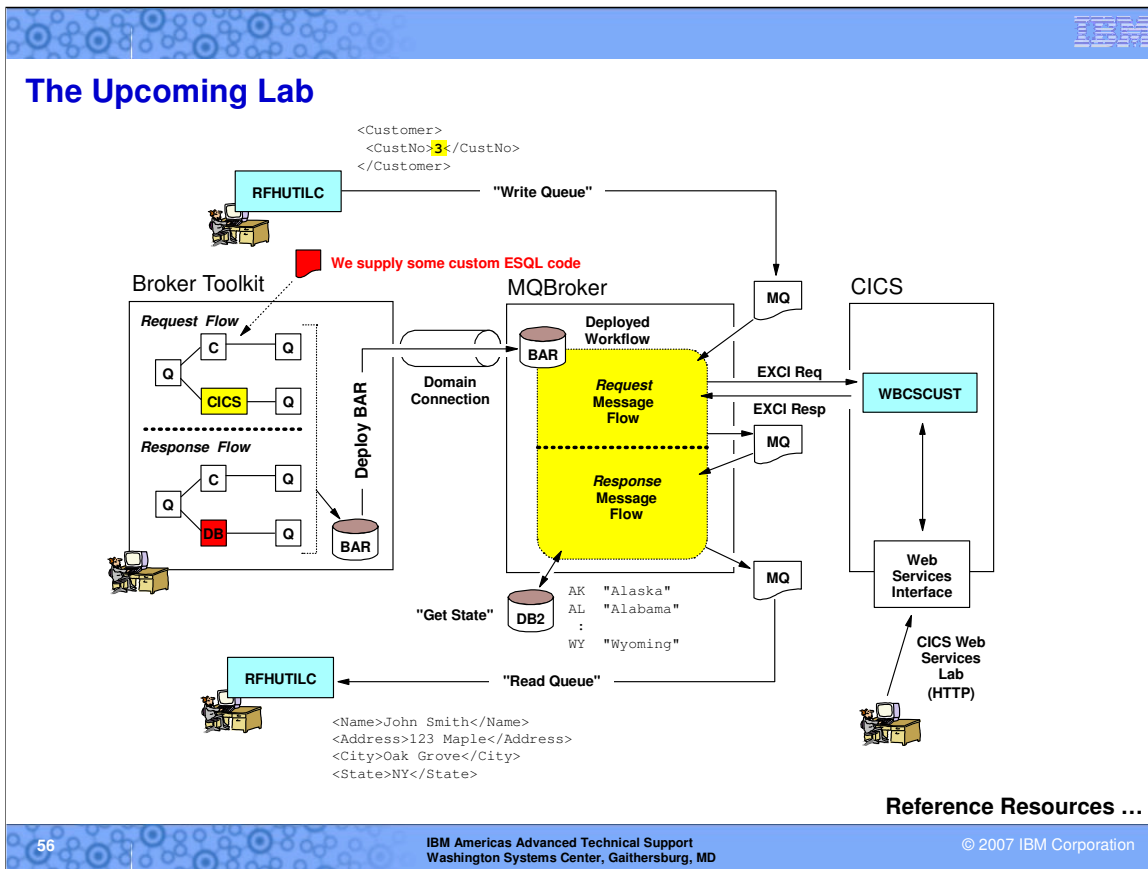
The picture above shows WMB acting as a kind of centralized hub for the broader ESB comprised of WESB. That's not a requirement ... it's just one suggested topology.

There is, of course, an *enormous* amount of technical details that would need to be fulfilled before such a topology could be fully operational. Nowhere in this presentation am I suggesting this stuff is pure magic. What this picture shows is how the product architecture permits a broader topology to be created.

So we see the bigger picture coming together, as the bullet points illustrate.

We've not yet really talked about WPS. That's the next unit in this workshop.





This picture is a schematic of the lab we'll do next. At the highest level, what this lab does is take in non-SOAP XML, convert it to COMMAREA and drive a CICS application. The result is then formatted back into XML, with a bit of data augmentation done with a Database node that fetches the full state name from the abbreviation that's returned from CICS ... for example, AZ is augmented to include the full name "Arizona."


We'll create the overall flow, which consists of two flows -- a request flow and a response flow -- in the Broker Toolkit. It's a long but not difficult point-and-click exercise. The result will be a BAR file, which we'll deploy into a running copy of WMB up on the z/OS system. Deployment occurs over a "Domain Connection," which is really a defined TCP link to WMB on z/OS. The flow uses a CICS Request node to access CICS and drive the WBCSCUST application, which is the same application you drove in the earlier lab using the CICS Web Services support.

We'll invoke this flow by putting a very simple XML message onto an MQ queue. We'll use the IH03 rfutilc utility to place the message on the queue. Broker will pick up the XML, parse it, convert it to COMMAREA, drive CICS and then place the resulting data in COMMAREA format into an intermediate queue. That's the end of the request flow.

The response flow then takes over. It picks up the COMMAREA data out of the queue, converts that to XML, then fetches the full name for the state that corresponds to the two-letter abbreviation returned from CICS. It does this with a database node. The updated XML is then placed on the output queue, which you'll read back using the rfutilc utility.

There is custom code in this mix. We're going to supply you with some ESQL which will provide the logic needed to build the COMMAREA data from received XML, and to format error and failure messages should they occur.





## Resources for WebSphere Message Broker

**WebSphere Message Broker Library:**  
<http://www.ibm.com/software/integration/wbimessagebroker/library/>

**WebSphere Message Broker V6 Info Center:**  
<http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp>

**WebSphere Message Broker SupportPacs:**  
<http://www.ibm.com/support/docview.wss?rs=171&uid=swg27007197#5>

**IBM Redbook – WebSphere Message Broker Basics :**  
<http://www.redbooks.ibm.com/abstracts/sg247137.html?Open>

**WebSphere Message Broker – System Requirements :**  
<http://www.ibm.com/software/integration/wbimessagebroker/requirements/>

## Resources for WSRR

**FAQs**  
<http://www.ibm.com/software/integration/wsrr/library/faqs.html#f1>

**WSRR InfoCenter**  
<http://publib.boulder.ibm.com/infocenter/sr/v6r0/index.jsp>

**More ...**

57

IBM Americas Advanced Technical Support  
Washington Systems Center, Gaithersburg, MD

© 2007 IBM Corporation

Reference material.

End of Unit