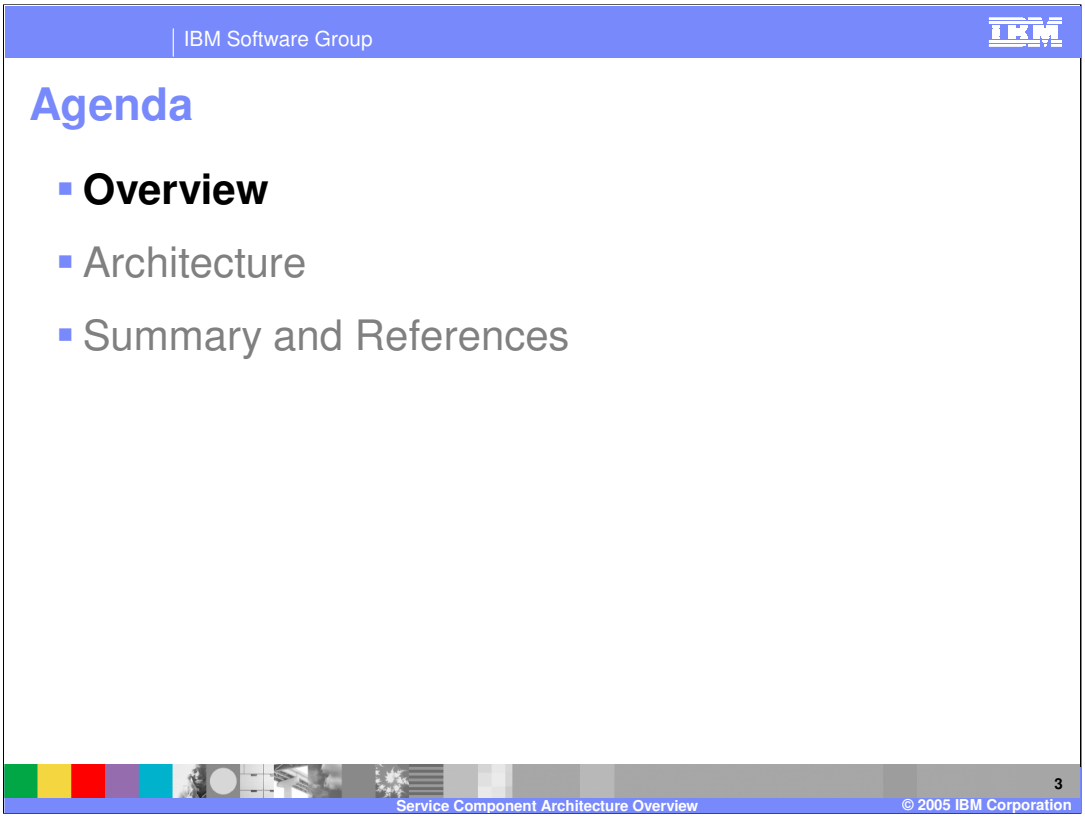
This presentation will provide an Overview of Service Component Architecture (SCA).

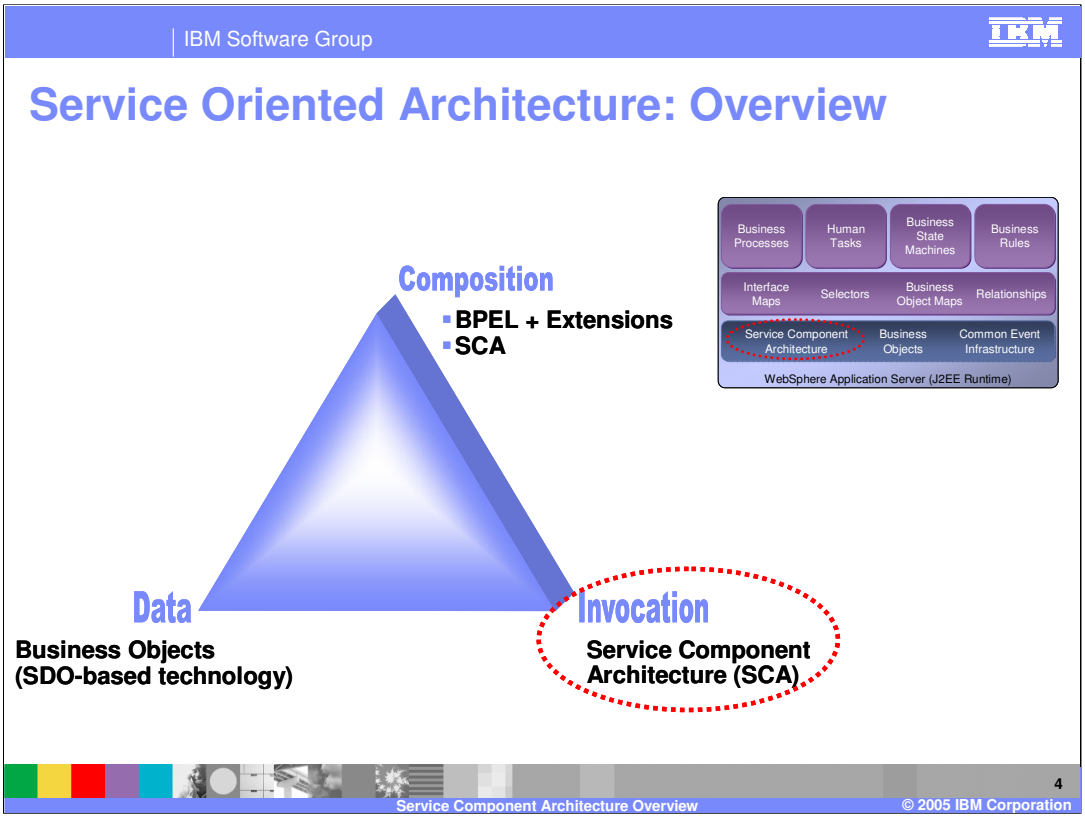
Goals

- Introduce Service Component Architecture (SCA)

The goal of this presentation is to introduce Service Component Architecture.



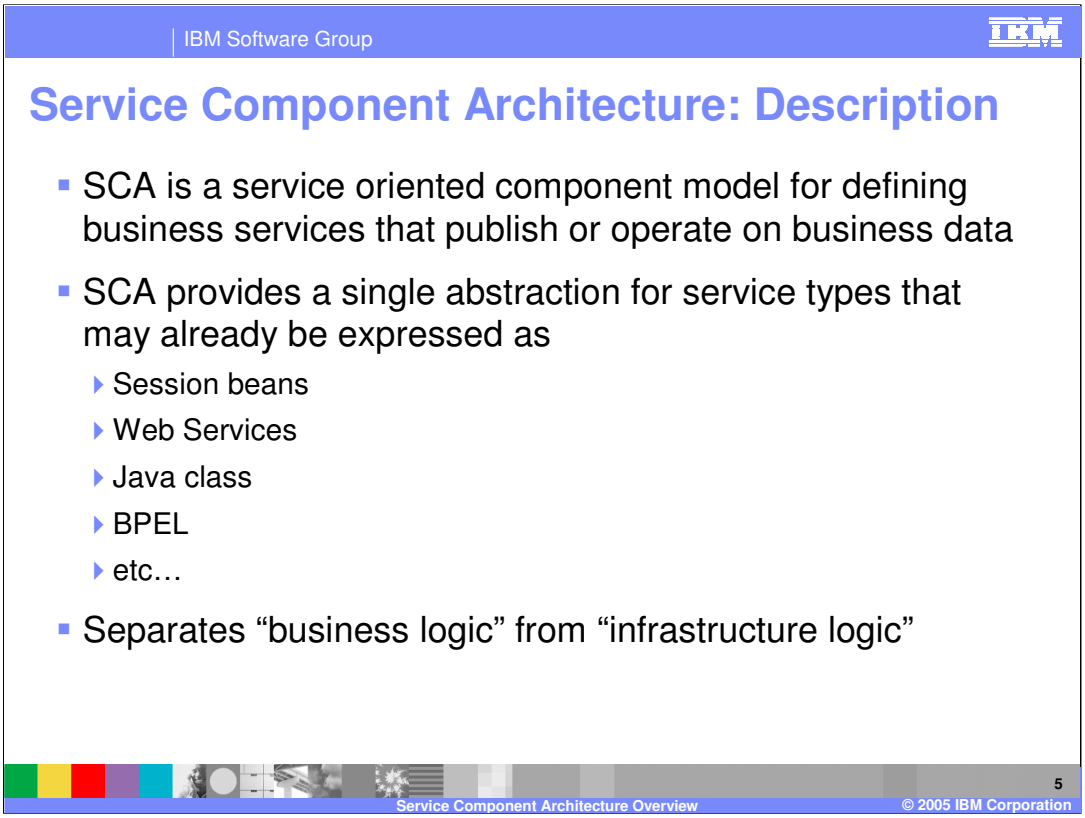
This section will provide an overview of Service Component Architecture (SCA).



The triangle of truth is a simple way to look at the important architectural constructs that make up a service oriented architecture. When you consider what is needed to build a service oriented architecture, the triad that makes up the triangle of truth quickly emerges. Specifically, there must be a way to represent the data that is exchanged between services, a mechanism for invoking services, and a way to compose services into a larger integrated business application.

Today there are many different programming models for supporting each construct in the triangle of truth. This situation presents developers with the challenge of not only solving a particular business problem, but also choosing and understanding the appropriate implementation technology. One of the important goals of the WebSphere Process Server V6 SOA solution is to mitigate these complexities by converging the various programming models used for implementing service oriented business applications into a simplified programming model.

This presentation focuses specifically on the Service Component Architecture (SCA) introduced in WebSphere Process Server V6 as the service oriented component model for defining and invoking business services. In addition to the important role SCA plays in providing an invocation model for the SOA solution in WebSphere Process Server V6, you will also learn in this presentation that it also plays a role in composing business services into composite business applications.

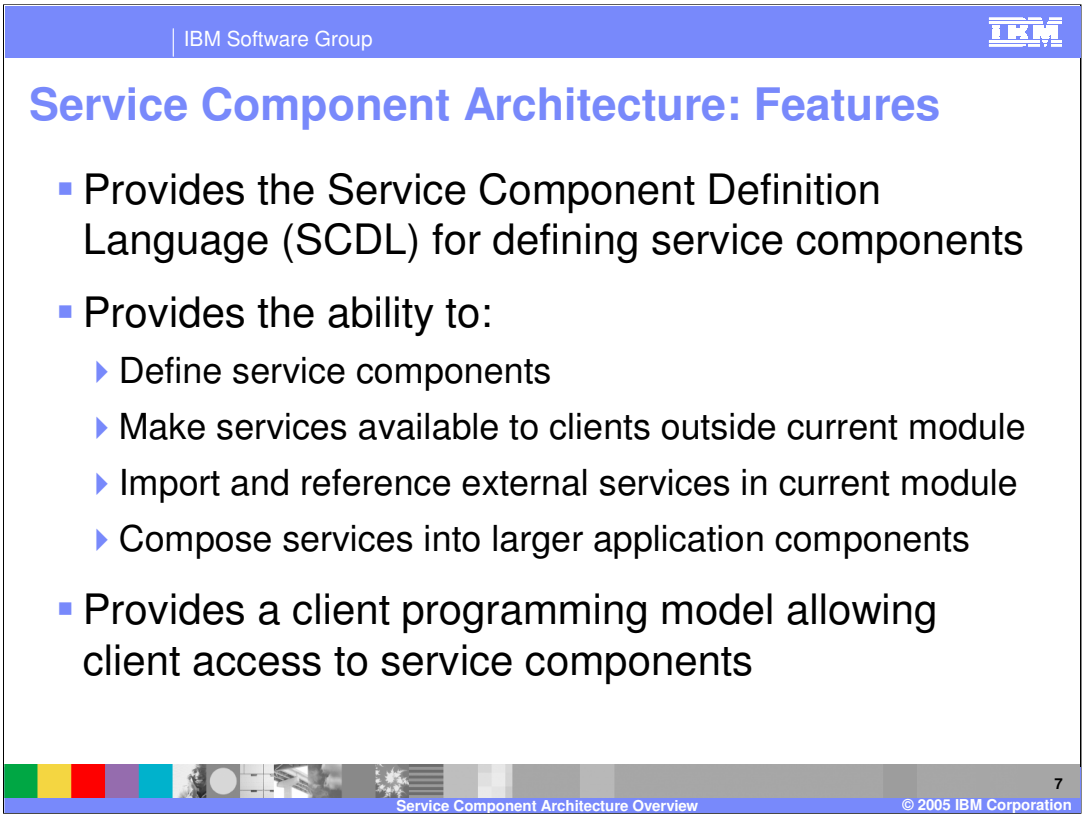


SCA is a service oriented component model for defining and invoking business services that publish or operate on business data. SCA is aimed at providing a simplified programming model for writing applications that run in a J2EE runtime environment, and is based upon concepts and techniques that are refinements of existing J2EE technology. One of the important aspects of SCA is to enable the separation between application business logic and the implementation details. In order to accomplish this, SCA provides a single abstraction for service types that might already be expressed as session beans, Web services, Java™ classes, or BPEL. The ability to separate business logic from infrastructure logic is important to help reduce the IT resources needed to build an enterprise application, and give developers more time to work on solving a particular business problem rather than focusing on the details of which implementation technology to use.

Agenda

- Overview
- **Architecture**
- Summary and References

This section will provide the architectural details of SCA.

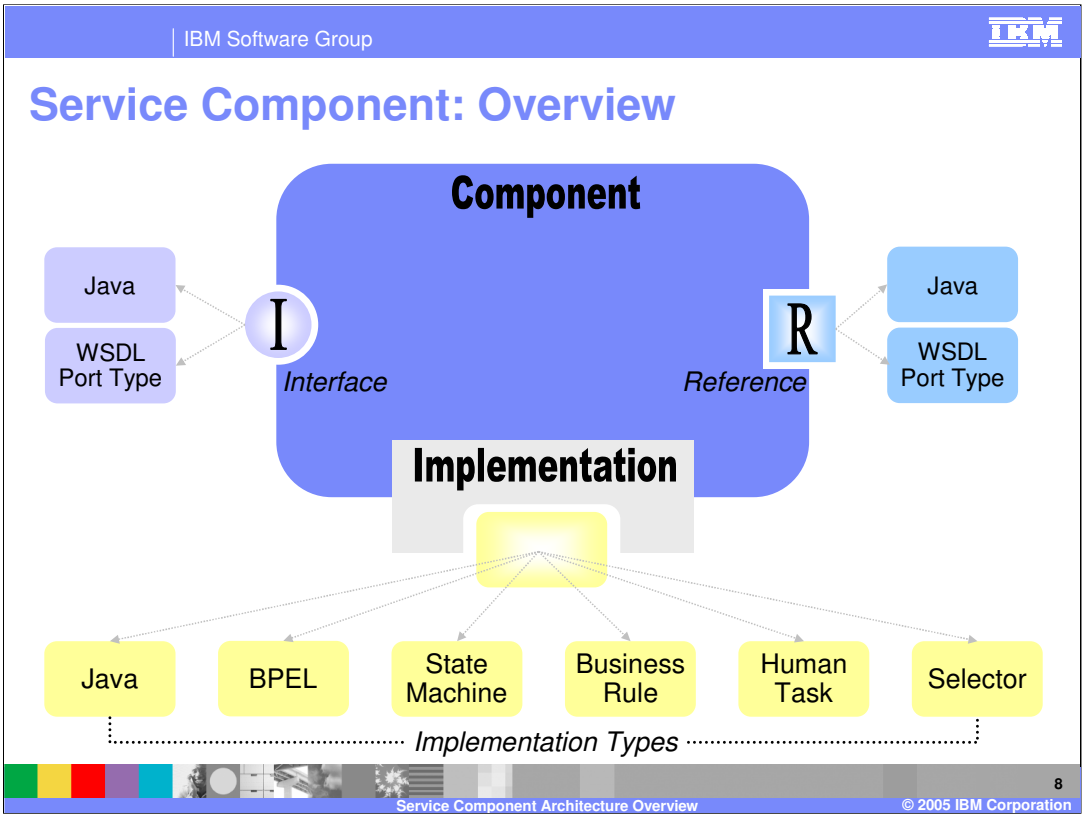


When learning a new technology or programming model, it is often useful to look at the pieces that compose the overall architecture of that technology. This slide lists some of the important features of SCA that you should be aware of as you begin learning about SCA.

First, the Service Component Definition Language (SCDL) provides the basis of SCA. SCDL is an XML based definition language used to define all SCA artifacts in a project. The WebSphere Integration Developer V6.0 tools support of SCA takes care of generating the appropriate SCDL definitions when building an SCA-based application. However, a basic familiarity with SCDL can certainly help you to understand the overall architecture and when debugging applications.

The next important part of SCA are the different types of artifacts that can be defined using SCDL. The various artifact types that exist in SCA were designed to support some of the basic requirements of this service oriented architecture. To start with, SCA needs a mechanism for defining a basic service component. Once there is a mechanism for defining service components, it is important to have the ability to make these services available to clients both inside or outside of the current SCA module. In addition to this, a construct designed to import and reference services external to the current SCA module must exist. Finally, SCA provides constructs for composing services and modules into larger applications. In the remaining slides of this section, you will learn about each of these SCA artifacts and how they can be composed into larger applications.

One final feature of SCA is the client programming model that allows clients to access and invoke service components in an SCA module. Later in this presentation, you will learn about the Java interfaces that are available for invoking services from a client.



The basic building block in SCA is the service component. The service component represents a business service that publishes or operates on business data. The diagram on this slide introduces the essential pieces of a service component definition.

Interfaces

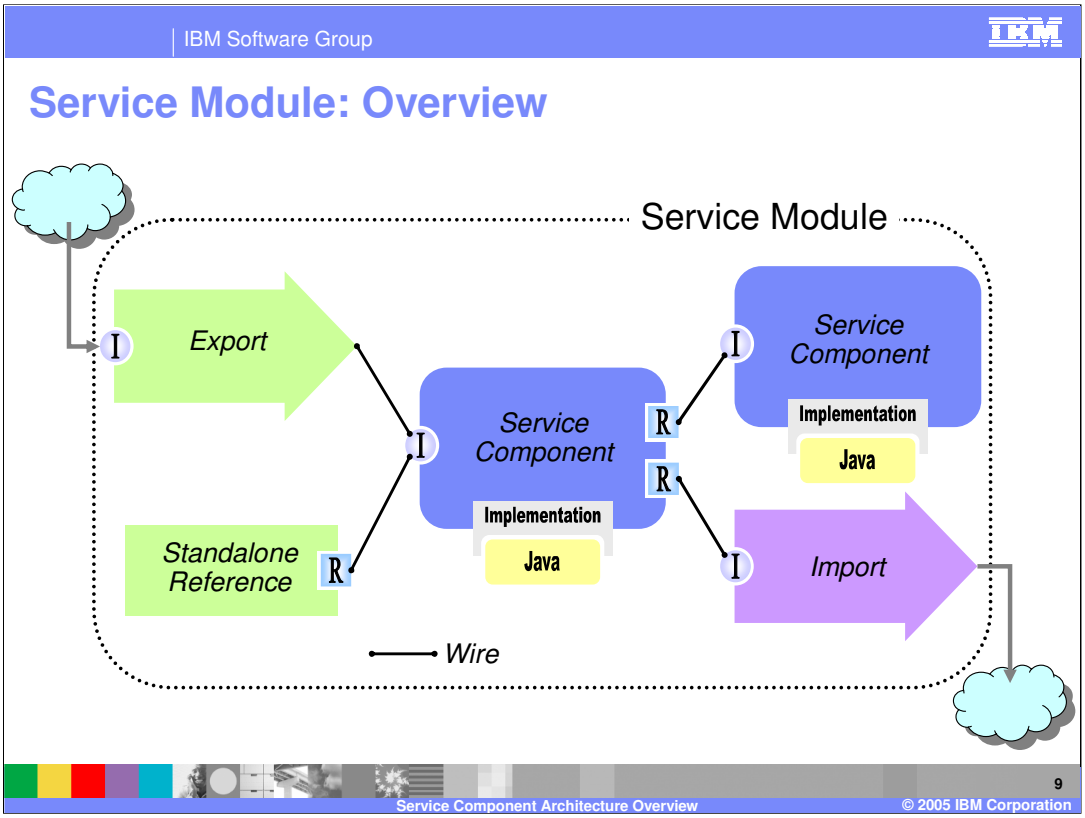
A service component has one or more interfaces with which it is associated. The interfaces associated with a service component advertise the business operations associated with this service. These interfaces can be specified as either Java interfaces or WSDL port type interfaces. However, you can not mix Java and WSDL port type interfaces on the same service component definition. The arguments and return types for these interfaces are specified as simple Java types, Java classes, Service DataObjects, or XML Schema (for WSDL port type interfaces).

Implementation

Also associated with a service component definition is an implementation. As the diagram indicates, there are multiple language types available for implementing a service component. This presentation will primarily focus on the Java implementation type. However other presentations are available to discuss the details of the other implementation types that are available.

References

Each service component can access other services in their implementation. For this, a service component definition can include zero or more references to other service components or imports included in the current module.



The previous slide introduced the service component as the basic building block in SCA. This slide provides a broader look at SCA and the other pieces that make up the architecture.

This discussion begins with the service module, which provides the basic unit of deployment and administration in an SCA-enabled runtime. A service module encapsulates the various artifacts available with SCA and is illustrated in the diagram on this slide. The following is a summary of the elements that make up a service module.

Service Component

A service module may have zero or more service components included with it. In order to access these services by a client (SCA or non-SCA) there must exist at least one reference to the service or the service must be exposed with an export.

Imports

A service module can have zero or more imports included with it. An import is used to access services that are outside the current SCA module. Once an import has been defined, other services from within the module can reference the imported service as if it was a regular service component defined in the module.

Exports

A service module can have zero or more exports included with it. An export is used to expose a particular service to clients outside the current SCA module.

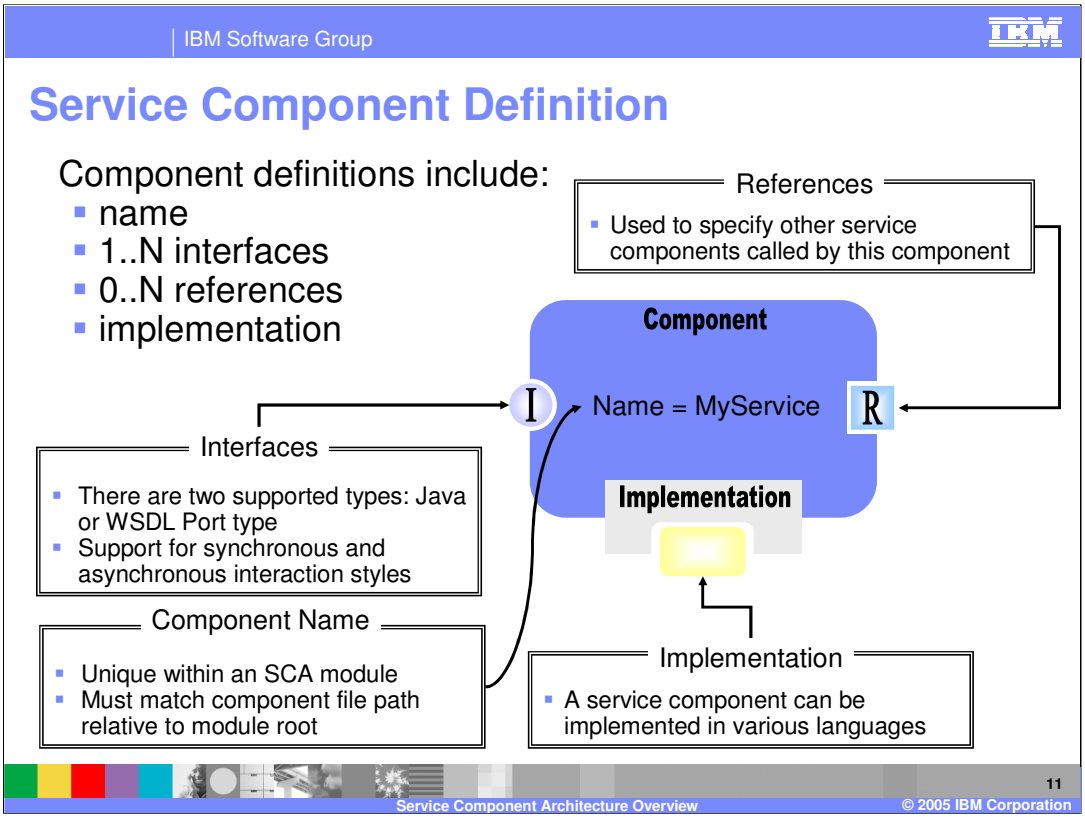
Stand-alone References

A service may include a stand-alone references file that includes references to services in the module that can be used by SCA and non-SCA services.

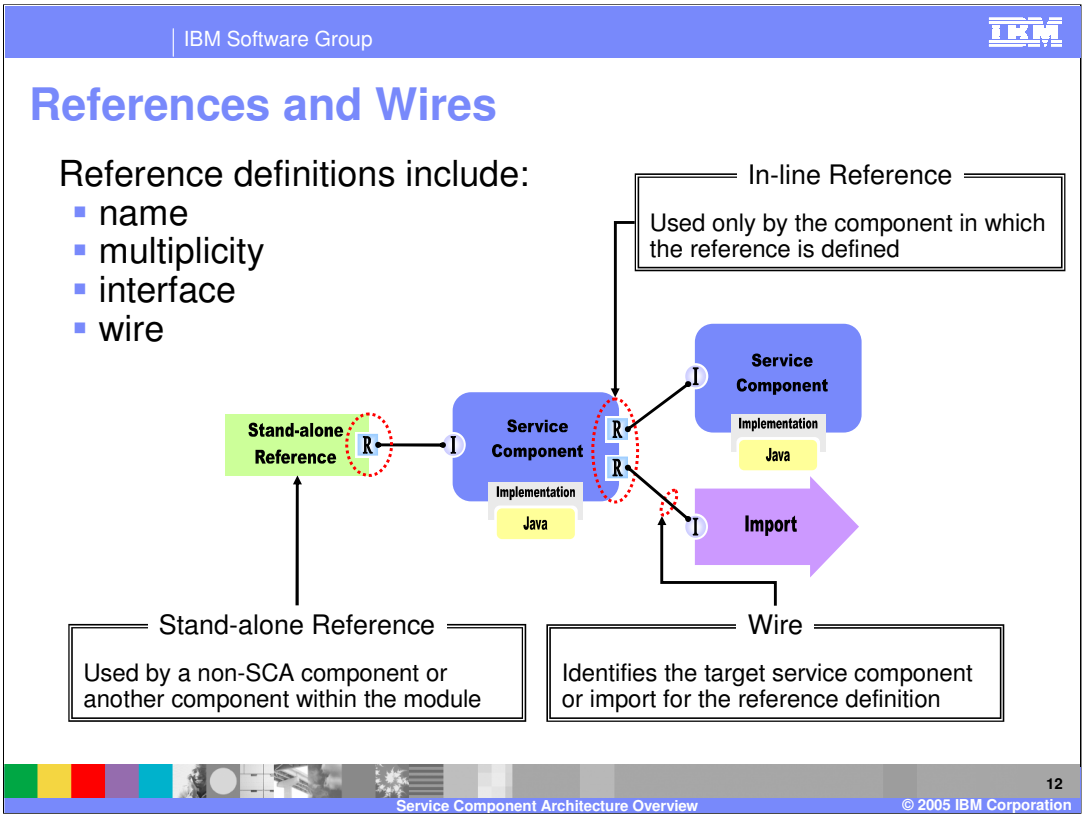
Service Module: Artifacts

Artifact	Comments
Module Definition	▪ Contained in the **sca.module** file at the root SCA project JAR
Service Components	▪ A module can contain 0..n service definitions ▪ Each component definition is contained in a **<SERVICE_NAME>.component** file
Imports	▪ A module can contain 0..n import definitions ▪ Each import definition is contained in a **<IMPORT_NAME>.import** file
Exports	▪ A module can contain 0..n export definitions ▪ Each export definition is contained in a **<EXPORT_NAME>.export** file
References	▪ Two types of References ‣ In-line (contained within a service component definition) ‣ Stand-alone ▪ Stand-alone references are defined in the **sca.references** file
Other Artifacts	▪ Other artifacts include: Java Classes, WSDL files, XSD files, BPEL, etc.

The table on this slide includes the primary artifacts that make up an SCA service module. Listed in the comments column is the name of the file for the artifact that includes the SCDL definition for that particular artifact type.

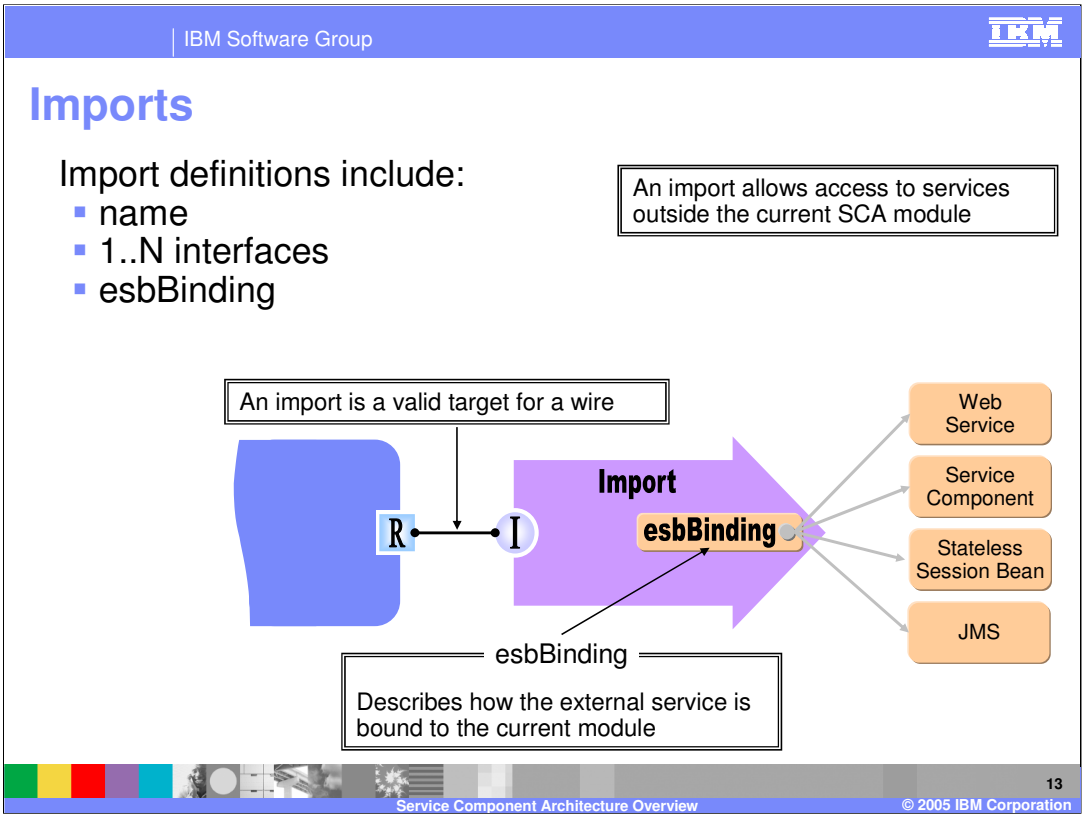


This slide provides a more detailed look at the service component definition introduced earlier in this presentation. Each service component must have a unique name within the SCA module and it must match the file path relative to the module root. As noted on the previous slide, the service component definition is included in a file called <SERVICE_NAME>.component. Next, each service component can have one or more interfaces associated with it, which can be either Java or WSDL port type interface definitions. The interfaces associated with a service component can support either a synchronous or asynchronous interaction style with clients calling the service. This feature is discussed in more detail in upcoming slides in this presentation. As noted earlier, each service component can be implemented in various ways, specified by the implementation definition. Finally, service components can invoke other service components or imports defined in the current service module. In this case, the appropriate reference must be defined to indicate which service is used. Often this type of reference is in-lined in the service component definition, although it may alternatively be placed in the stand-alone references file. Each service component definition can have zero or more references to other services called by the service component being defined.



SCA and non-SCA clients calling a service component need a reference to that service in order to invoke it. This slide reveals some of the details of defining references. Each reference has a name, used to look up the appropriate service by a client using the client programming model. Details of this look up code will be covered later in this presentation. In addition to the name, a reference also includes an interface element. The multiplicity for a reference indicates how many wire definitions can name this reference as the source. Finally, the wire definition specifies the name of the target service component or Import that will resolve the reference.

There are two ways to define references. The first way is to in-line the reference in the service component definition. Using this approach, the references are only available to the service component in which the references are included. Another approach is to include reference definitions within the stand-alone references file. For this approach, the references can be used by a non-SCA client or by another component within the module. An example of a non-SCA component that may use a reference in the stand-alone references file is a user interface component such as a JSP that needs the ability to invoke a particular service. In order to invoke a service component, the client needs a reference so that it can use the SCA runtime to lookup the appropriate service to invoke.



SCA import components allow clients in an SCA module to access services that are outside the current SCA module. Like service components, imports have a name and also a set of 1..N interfaces with which they are associated. Imports also have a binding attribute, which is used to describe how the external service is bound to the current module. The common binding types are indicated on this slide, and the following is a summary of each type:

Web Service Imports allow clients to access external Web services using the SCA programming model.

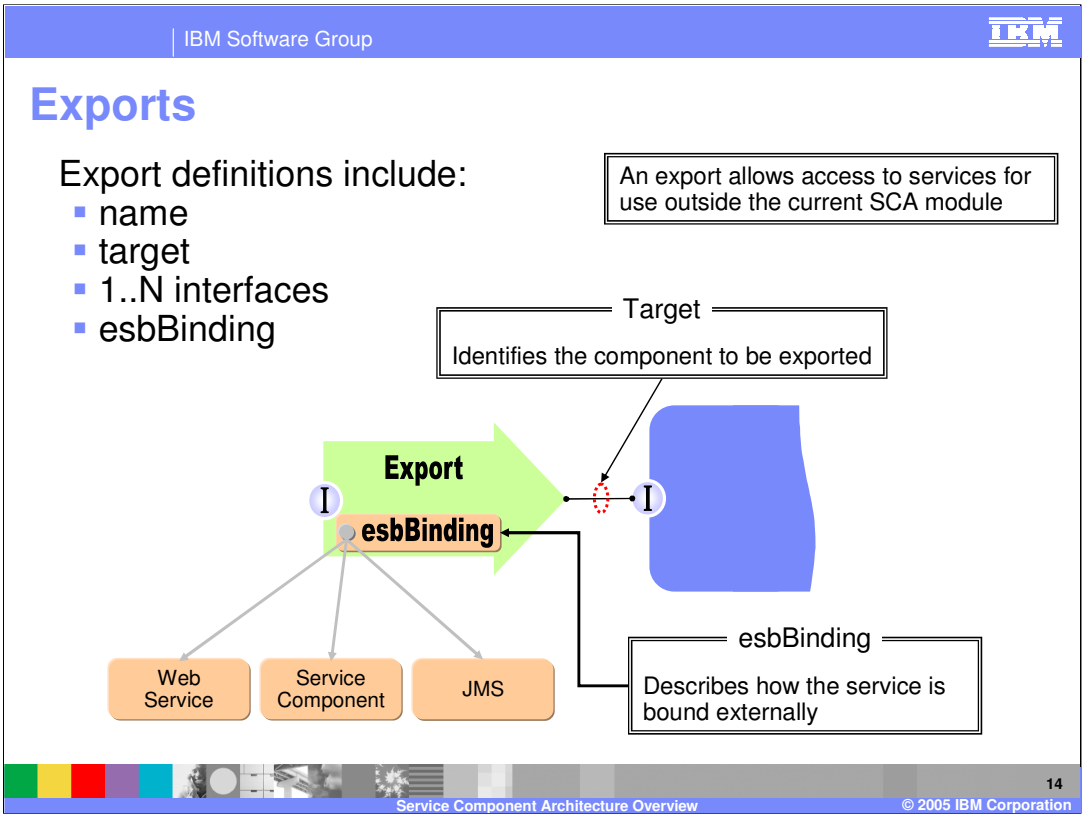
Service Component Imports allow clients to access other SCA services that reside in another SCA module and have been exported using an SCA export binding type.

Stateless Session Bean Imports allow clients to access services, that have been exposed using a stateless session bean using the SCA client programming model.

JMS Imports allow clients access to services that are available using a JMS protocol.

There is also an EIS binding type that is used in association with adapter components. However, a discussion of this binding type is outside the scope of this presentation.

Imports can be thought of as a special type of service component in an SCA module. Imports are valid targets in a wire definition for a service reference. This means that to a client invoking a target service the client programming model is the same whether the reference points to an import or another service component.



SCA exports provide access to service components defined in an SCA module for use by clients outside of the current SCA module. Export components include a name and a target attribute, which names the service component that is to be exported. Like import components, exports have a binding attribute that indicates how the service is bound externally. The common binding types are indicated on this slide, and the following is a summary of each type:

The Web Service binding type allows service components to be exported and made available to external clients as a Web service.

The Service Component binding type allows service components to be exported to other SCA clients in modules external to the current SCA module. This binding type is used in conjunction with a corresponding import component with an SCA binding type in another SCA module.

The JMS binding type allows service components to be accessed externally using a JMS protocol.

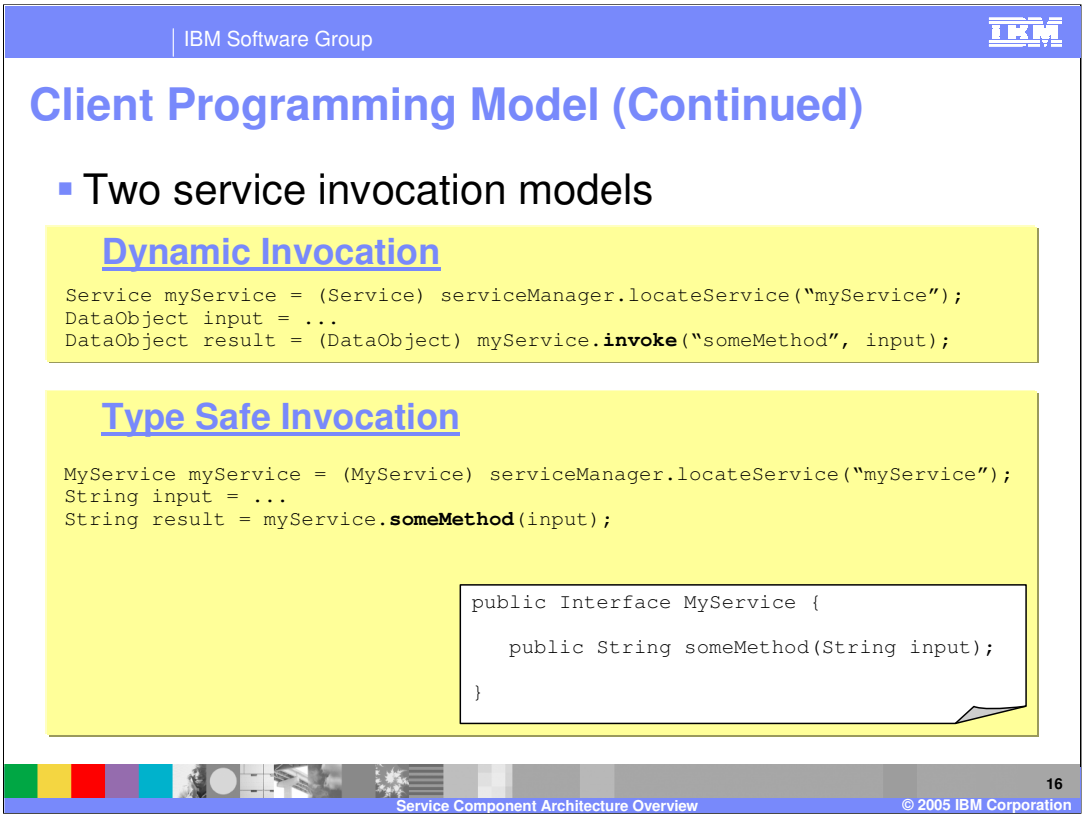
There is also an EIS binding type for exports that is used in association with adapter components. However, a discussion of this binding type is outside the scope of this presentation.

Client Programming Model

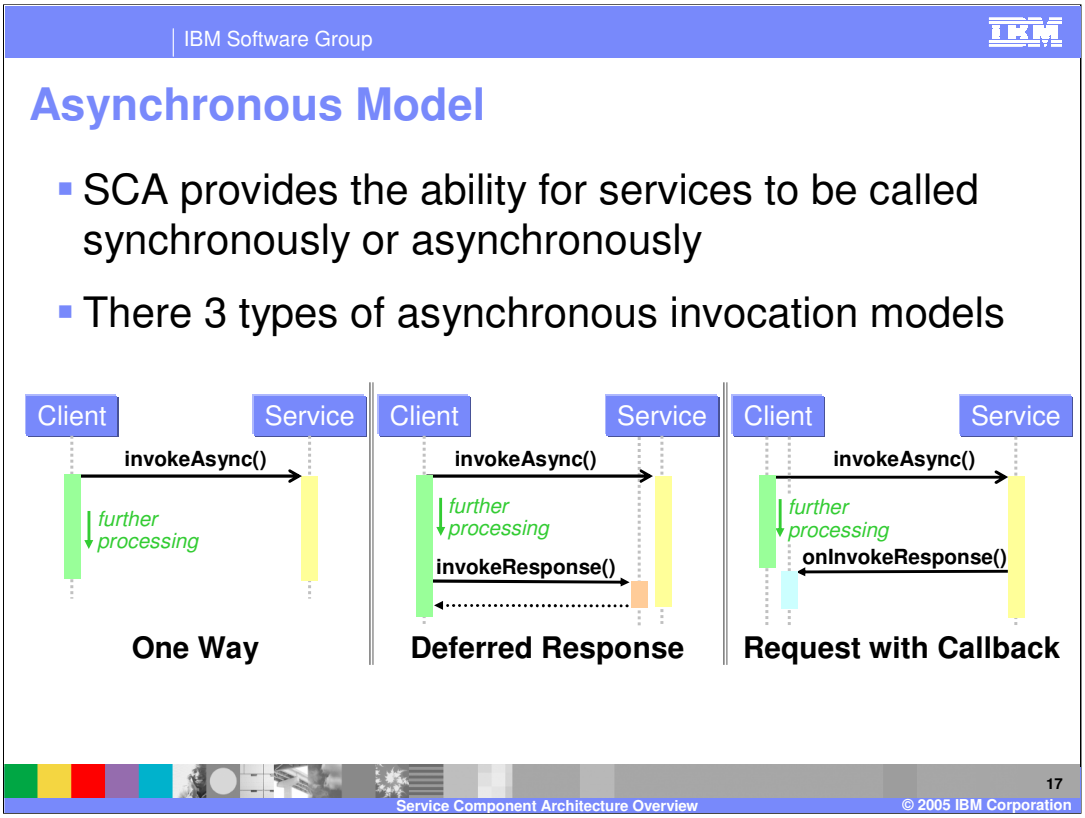
- Client programming model allows clients to
  - Locate services
  - Invoke methods on services
- Clients locate services with the ServiceManager
  - Key class is
    - `com.ibm.websphere.sca.ServiceManager`
  - Two ways to instantiate a ServiceManager depending on required lookup scope for service
  - Use the following method to locate a service
    - `com.ibm.websphere.sca.Service locateService(String);`

The SCA client programming model provides two primary functions for clients. The programming model exposes an interface that allows clients to locate services within the current module, and once a service is located the client programming model provides a way for the client to invoke operations on that service.

The key interface that clients should be aware of for locating services is `com.ibm.websphere.sca.ServiceManager`. This interface includes a locateService method that returns a reference to the service implementation for the service requested. The string parameter that is passed into the locateService method represents the reference name for the service that the client wishes to locate. The Java documentation for the SCA programming model is included in the WebSphere Process Server V6.0 information center, and is also included if you choose to install the Java documentation as part of the WebSphere Process Server V6.0 installation.



Once a client has located the appropriate service, there are two types of invocation models that can be used to make a call to an operation or method offered by the service. First, there is a dynamic invocation style of interaction. The key interface for this style of interaction is com.ibm.websphere.sca.Service. The invoke() method on this interface takes the name of the operation that you wish to invoke, along with the parameters needed to call that operation. Clients can also use a type safe invocation method to call a particular operation associated with a service. This type of invocation only works for interface definitions that are specified as Java. In this situation, the client casts the return from the locateService() call to the appropriate interface and can proceed calling the appropriate type safe method calls on that interface.



So far, this presentation has focused on the synchronous invocation model. However, SCA provides the ability for services to be called either synchronously or asynchronously. The next several slides will present more about the asynchronous programming model.

With asynchronous invocation in SCA, there are three types of asynchronous interaction styles available. With all three types of asynchronous invocation, the client receives control back immediately from the SCA runtime upon an invokeAsync() call. However, there are 3 different ways that the client can capture the response at a later time. First, the client can choose to discard the response entirely or if it is a call to a void method. In this case, the asynchronous invocation is said to be “one way”. Another option is for the client to call invokeAsync() and then continue processing until some later time when the client makes a request to capture the response. This scenario is termed “deferred response”. Finally, the client also has the option of doing an asynchronous “request with callback”. To do this, the client must first implement the ServiceCallback interface. Then, after making a call to invokeAsync(), the SCA runtime provides a callback to the ServiceCallback handler to provide the response to the client.

The next slide provides a summary of the interfaces needed to support these 3 asynchronous invocation models for the dynamic interaction style.

Dynamic Client Invocation

Interface	Methods	Description
Service	Object invoke(String, Object)	Used to invoke synchronous service requests
	Ticket invokeAsync(String, Object)	Used to invoke **one-way** or **deferred response** asynchronous service requests
	Ticket invokeAsyncWithCallback(String, Object)	Used to invoke **request with callback** asynchronous service requests. The client must implement the ServiceCallback interface
	Object invokeResponse(Ticket, long)	Used to get response in the case of **deferred response** invocation
ServiceCallback	void onInvokeResponse(Ticket, Object, Exception)	Callback interface must be implemented by the client using a **request with callback** asynchronous service invocation

This slide provides a summary of some of the key methods and interfaces needed to support both synchronous and asynchronous interaction when using dynamic client invocation. The Java documentation for these interfaces is included with the WebSphere Process Server V6.0 installation under <WPS_HOME>\web\apidoc), as well as in the product information center.

SCA Interactions

Interface Type	Invocation Model				Invocation Methods	
	Synchronous	One Way	Deferred Response	Request with Callback	Dynamic	Type Safe
WSDL Port Type	●	■	■	■	YES	NO
Java Interface	●	■	■	■	YES	YES

● Data passed by reference in the same SCA Module

■ Data passed by value

The table shown on this slide lists the various invocation models and the method by which the data is passed, whether by reference or by value. For synchronous invocation, data is passed by reference within the same SCA module, while for asynchronous calls the data is passed by value. The table on this slide also summarizes when it is possible to use either type safe or dynamic invocation based upon the interface type. The dynamic invocation methods are always available for either WSDL port type or Java interfaces. However, in order to have type safe invocation methods available to the client a Java interface type must be used for the interface definition on the appropriate client reference.

SCA Quality of Service

- Qualifiers provide the ability to specify quality of service requirements from the SCA runtime
- Qualifiers fall into one of the following categories
  - Transaction
  - Activity Session
  - Security
  - Asynchronous Reliability
- Qualifier can be specified on component references, interfaces, and implementations

Qualifiers are an important part of SCA because they allow developers to place quality of service requirements on the SCA runtime. There are several different categories of qualifiers available in SCA. These categories of qualifiers are transaction, activity session, security, and asynchronous reliability. Each SCA qualifier has a particular scope within the SCDL definition for a component where the qualifier can be specified. For example, some qualifiers can be specified at the references level, while others may only be valid at the interfaces or implementation level. The following slides will list the various qualifiers that are available and the valid scope for each. The qualifiers are sorted by the type of quality of service they provide, such as transaction or security.

Qualifiers

Type	Qualifier	Scope	Description
Transaction	**transaction**	Implementation	**global** – A global transaction must be present to run the component **local** – A global transaction must not exist to run the component **any** – Component is unaffected by transactional state
	joinTransaction	Interface	**true** – Hosting container joins client transaction **false** (default) – Hosting container will not join client transaction
	suspendTransaction	Reference	**true** – Synchronous invocations of target component do not run within client global transaction. **false** – Synchronous invocations of target component run within client global transaction
	deliverAsyncAt	Reference	**call** – Asynchronous invocations of a target service occur immediately **commit** – Asynchronous invocations of a target service occur as part of a global transaction

Transaction qualifiers allow developers to request a particular transactional environment for the service components in an SCA module. The following is a summary of these qualifiers:

The joinTransaction qualifier is set at the interface scope of a service component. This qualifier can be set to either true or false, false being the default. If set to true, it instructs the runtime not to suspend a global transaction (if present) at the interface boundary. If set to false, it instructs the runtime to suspend a global transaction (if present) at the interface boundary. Exposing the joinTransaction transactional qualifier on an interface provides metadata that can be used by assemblers and deployers to ensure that the assembled application will behave as required. It is up to the assembler and deployer as well as dynamic runtimes to reason about whether or not a target component will federate with a propagated transaction.

The transaction qualifier is set at the implementation scope of n service component. This qualifier can be set to either 'global', 'local' (default), or 'any'. When set to global, the component will run in the context of a global transaction. If a global transaction is present on invocation, the component will be added to this global transaction scope. If set to local, the component will run in the context of a local transaction. Finally, if the value is set to any then if a global transaction is present the component will join the current global transaction scope. However, if a global transaction is not present, the component will run in the context of a local transaction.

The suspendTransaction qualifier is set at the reference level of a service component and identifies whether or not a global transaction should be suspended before invoking the target service associated with the reference. This qualifier can be set to either true or false (default).

The deliveryAsyncAt qualifier is similar to the suspendTransaction qualifier, except that it pertains to asynchronous interactions rather than synchronous types as is the case with suspendTransaction. The deliverAsyncAt qualifier can have the value of call (default) or commit. If set to call, it indicates to the runtime that the message for the asynchronous interaction should be committed to the queue immediately when the call has been made. The value of commit on the other hand indicates that the message should be committed to the queue as part of a transaction associated with the current unit of work.

Qualifiers (Continued)

Type	Qualifier	Scope	Description
Asynchronous Response	**reliability**	Reference	Specifies the quality of service level for asynchronous message delivery. Reliability can be one of the following values: **bestEffort** or **assured**
	requestExpiration	Reference	Specifies the length of time (milliseconds) after which an asynchronous request is to be discarded if not delivered
	responseExpiration	Reference	Specifies the duration (milliseconds) between the time a request is sent and the time a response or callback is received
Security	**securityIdentity**	Implementation	The **permission** specifies a logical name for the identity under which the implementation executes at runtime.
	securityPermission	Interfaces, Interface, Method	The caller identity must have the **role** specified from this qualifier in order to have permission to run the interface or method

There are three qualifiers available for indicating quality of service for asynchronous response. Each of the asynchronous response qualifiers are specified at the reference scope. The following is a summary of asynchronous response qualifiers:

- The reliability qualifier is used to specify the quality of service level for asynchronous message delivery. The reliability can be set to either bestEffort or assured (default).

- The requestExpiration qualifier is used to specify the length of time the runtime should hold onto a asynchronous request if it has not yet been delivered. After the time indicated for this qualifier, given in milliseconds, this request is discarded.

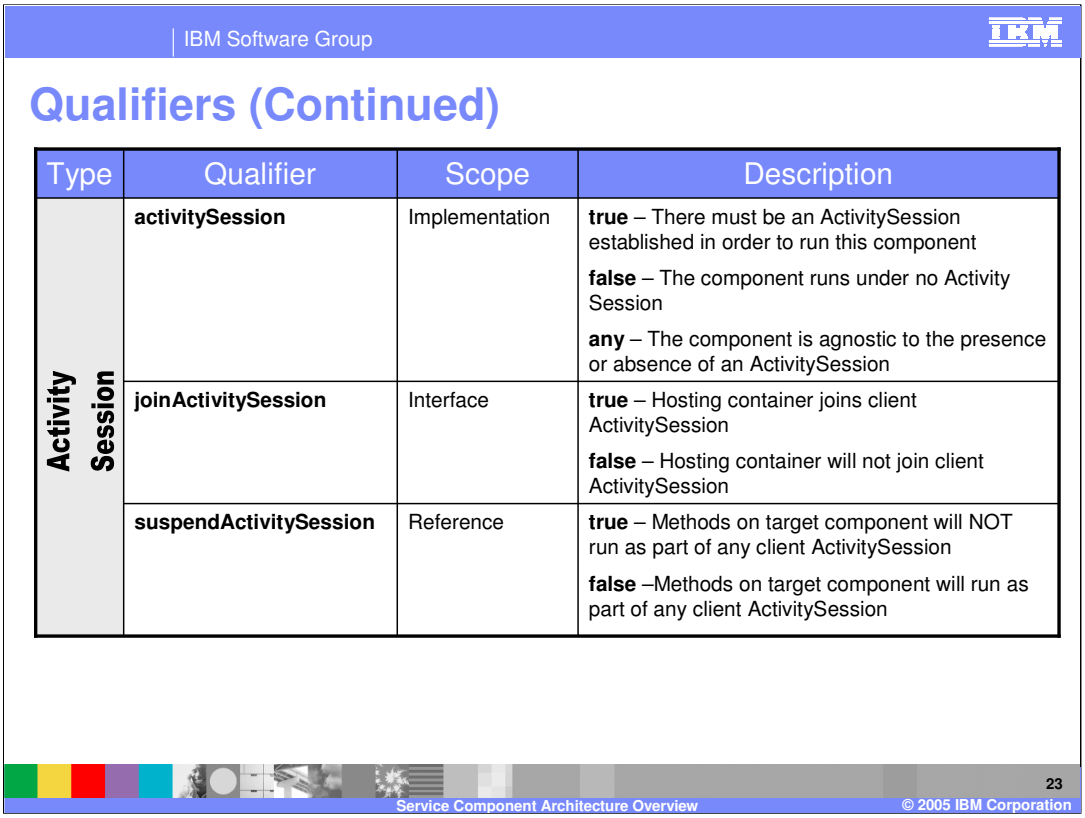
- The responseExpiration qualifier is used to specify the length of time that the runtime must retain an asynchronous response or must provide a callback. The value for this qualifier is given in milliseconds.

There are two qualifiers available for indicating quality of service related to security. The following is a summary of these qualifiers:

- The securityIdentity qualifier is used to specify the security identity under which the implementation for the service component should run at runtime. This qualifier must be placed at the implementation scope for the service component and the value given must match the logical name for the identity under which the component will run.

- The securityPermission qualifier is specified at the interfaces level, including interface, or method level. The value for this qualifier indicates that a caller of this service must have the role that is specified in order to invoke the service.

For both the securityPermission and the securityIndentity, the underlying implemenation for these qualifiers is based on existing J2EE concepts.



The set of activity session qualifiers are similar to the transaction qualifiers introduced earlier. The ActivitySession service is a Websphere programming model extension that can provide an alternative unit of work when compared with global transactions. In fact, an activity session context can be longer lived than a global transaction and can even include global transactions. The following is a summary of the activity session qualifiers:

The joinActivitySession qualifier is set at the interface level, and indicates whether or not the component should join the activity session of a client caller. There are two values for this qualifier, true and false (default). If set to true it indicates that the runtime should not suspend an activity session if present when the component is invoked. If set to false it indicates that the an activity session should be suspended before invoking the component.

The activitySession qualifier is specified at the implementation level and is used to indicate whether or not an activity session should or should not exist in order to run the service component with which it is associated. This qualifier can be set to either ‘true', ‘false', or 'any‘ (default). If set to true, it indicates that the component will run as part of an activity session. If set to false, the component should not run as part of an activity session. This means that the joinActivitySession should also be set to false for any interfaces specified for the component. Finally, if this qualifier is set to any, the component will run as part of an activity session if it is present, otherwise it will not.

The suspendActivitySession qualifier is set at the reference level and is used to indicate whether or not a target service associated with a reference will get called as part of the calling activity session or not. If set to true, the activity session is suspended and the methods on the target component will not run as part of the client activity session. If set to false (default) the activity session is not suspended and methods on the target component will run as part of the client ActivitySession.

Agenda

- Overview
- Architecture
- **Summary and References**

This section will provide a summary of SCA.

Summary

- SCA is the fundamental component model for WebSphere Process Server V6
  - Programming model the Service Oriented Architecture solution
- SCA helps separate business logic from implementation
  - Focus is on assembling solutions rather than implementation details
  - Mitigates need for integration developers to have deep knowledge of Java or J2EE
  - Aimed at helping J2EE developers become more productive

SCA is the fundamental component model for WebSphere Process Server V6 and provides the basis of the service oriented architecture solution. SCA helps separate business logic from implementation and allows developers to focus on assembling solutions rather than implementation details.

References

- Service Data Objects (SDO)
  - http://www-128.ibm.com/developerworks/library/j-commonj-sdowmt/
  - http://www-106.ibm.com/developerworks/java/library/j-sdo/
  - http://download.eclipse.org/tools/emf/sdo/javadoc/commonj/sdo/package-summary.html

Section

Appendix

Type Safe Client Invocation

Interface	Methods	Description
HelloWorld	String sendMessage(String)	Used to invoke synchronous service requests to sendMessage
HelloWorldAsync	Ticket sendMessageAsync(String)	Used to invoke **deferred response** asynchronous service requests
	Ticket sendMessageAsyncWithCallback(String)	Used to invoke **request with callback** asynchronous service requests
	String sendMessageResponse(Ticket, long)	Used to get response in the case of **deferred response** invocation
HelloWorldCallback	void onSendMessageResponse(Ticket, String Exception)	Callback interface must be implemented by the client using a **request with callback** asynchronous service invocation
HelloWorldImplAsync	void sendMessageAsync(String, HelloWorldCallback, Ticket)	Asynchronous HelloWorld implementations must implement this interface

This slide is provided for your reference to give a summary of the Interfaces that must be generated to support asynchronous invocation when the interface definition is of type Java and the type safe interaction style is used. In the tools environment with WebSphere Integration Developer V6.0, the interfaces listed in rows 2-4 are generated by the tools based upon the initial HelloWorld interface.

Trademarks, Copyrights, and Disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	CICS	IMS	MQSeries	Tivoli
IBM(logo)	Cloudscape	Informix	OS/390	WebSphere
e(logo)business	DB2	iSeries	OS/400	xSeries
AIX	DB2 Universal Database	Lotus	pSeries	zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds.

Other company, product and service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements and/or changes in the product(s) and/or program(s) described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (e.g., IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.