Web Services Development Concepts (WSDC 1.0)

May 2001

By Peter Brittenham
IBM Software Group

Notices

Contents

Figures

Tables

Preface

Abstract

This paper describes the approach for developing Web Services from the point of view of the developer of service providers and service requestors. The development approach explains the components, interactions, application development patterns and tools necessary to implement Web Services in general. It relates these common concepts to their application using Java™ technology.

The development approach presented in this paper includes high-level descriptions of the components and functions required for Web Services, and requirements on the tools and middleware to implement these components and functions. Some of this functionality exists today in products such as the IBM XML and Web Services Development Environment, the IBM Web Services Toolkit, and IBM WebSphere® Application Server. These and other products will implement additional functions in the future. However, the presence of a component, function or requirement in this paper does not guarantee that it will be implemented in future IBM products.

Target Audience

- Early adopters and implementers of Web Services
- External technical reviewers seriously evaluating the IBM Web Services approach. Reviewers should read the *Web Services Overview and Value* paper that explains the value of Web Services, and the *Web Services Conceptual Architecture* paper that explains the IBM approach to Web Services technologies.

Comments

Please send any feedback, technical or editorial, to Web Services at wbservcs@us.ibm.com or webservices/Raleigh/IBM@IBMUS.

Overview

When developing Web Services, there are build, deployment and runtime considerations for Web service developers and Web service requestors. This paper outlines the architectural components, subsystems and best practices for the creation, packaging and usage of a Web service.

The service requestor, service provider and service registry roles are incorporated in the development approach at varying levels depending on business needs and the robustness required to facilitate the publish, find and bind operations.

Web Services Development Lifecycle

As described in the *Web Services Conceptual Architecture* paper, Web Services have three primary component roles: service registry, service provider and service requestor.

Each component role has specific requirements for each phase of the development lifecycle. This paper describes the interactions between the Web Services component roles of service provider and service requestor. (The development and deployment of a service registry is outside the scope of this paper.) This paper describes the primary tasks defined by the development lifecycle. The service provider and service requestor roles are logical constructs, because a single service can exhibit characteristics of both a service requestor and service provider.

An example of a complete end-to-end lifecycle scenario would start with the creation and publication of a service interface (build), proceed to the creation and deployment of the Web service (deploy), move on to the publication of the service implementation definition and end with the invocation of the Web service by the service requestor (run).

The development lifecycle encompasses the following phases: build, deploy, run, and manage.

Build

The build phase of the lifecycle includes development and testing of the Web Services implementation, the definition of the service interface description and the definition of the service implementation description. Locating an existing service interface definition is also a build-time function. The Web Services implementations can be provided by creating new Web Services, transforming existing applications into Web Services, and composing new Web Services from other Web Services and applications. This paper focuses on the development concepts for a single Web service.

Developing a new Web service involves using the programming languages and models that are appropriate for the service provider’s environment. Transforming existing applications into Web Services involves generating service interfaces and service wrappers to expose the application’s relevant business functions. Composing new Web Services from existing Web Services involves sequencing and orchestrating message flows between programs directly or with workflow technologies. The Web Services that are used to compose a workflow can exist within the enterprise and outside the enterprise.

There are some similarities between a Web service development approach and object-oriented programming, because it uses constructs such as encapsulation, interface inheritance using *tModels,* and dynamic binding. This means that object-oriented design methodologies can be applied to Web Services design, but it is not required to design a Web service.

Deploy

The tasks associated with the deploy phase of the development lifecycle include the publication of the service interface and service implementation definition, deployment of the runtime code for the Web service and integration with back-end legacy systems.

For Web Services representing transformed applications, deployment can include only the Web service wrapper because the application might already be deployed. For service flows, deployment will include customization of the workflow manager and business process manager to execute and monitor the new flows. Additional administrative integration into the execution environment would also have to be performed for the definition of operation-based authorization and service credentials and integration with back-end legacy applications.

Run

During the run lifecycle phase, the Web service is fully deployed and operational. During this stage of the lifecycle, a service requestor can find the service definition and invoke all defined service operations. The runtime functions include static and dynamic binding, service interactions as a function of Simple Object Access Protocol (SOAP) messaging and interactions with legacy systems.

Manage

The manage phase of the Web service lifecycle covers ongoing management and administration of the Web service application. Security, availability, performance, quality of service and business processes must all be addressed. Because this paper focuses on the development of Web Services, this phase of the lifecycle is not covered.

Developing Web Services

This section describes the Web service lifecycle for each Web service component role: service registry, service provider and service requestor.

Service Registry

The service registry provides a role in the Web Services development approach, but it is a passive participant. It is assumed that the registry has been built and deployed before it is selected for use by the service provider or service requestor. For this reason, the development lifecycle for the service registry is not provided in this paper.

Service Provider

There are four basic methods that a service provider can use to develop a Web service. The method that is used is based on the existence of the service interface and application that will become the Web service. Table 1 provides an overview of these development methods.

	New Service Interface	**Existing Service Interface**
New Web Service	green field	top-down
Existing Application	bottom-up	meet-in-the-middle

Table 1 .Service provider methods

Each of these methods for developing a Web service is described in detail below. When describing the tasks within each phase of the lifecycle, some tasks are repeated between methods. The tasks that are repeated are described in detail only when they are listed for the first time.

Green Field

Service Interface	New
Web Service	New

Table 2. Green field method

As shown in Table 2, the green field method for developing Web Services describes how a *new* service interface will be created for a *new* Web service. Figure 1 illustrates how using this method, the Web service is created first, and then the service interface definition is generated from the new Web service. The service interface and service implementation are both owned by the service provider.

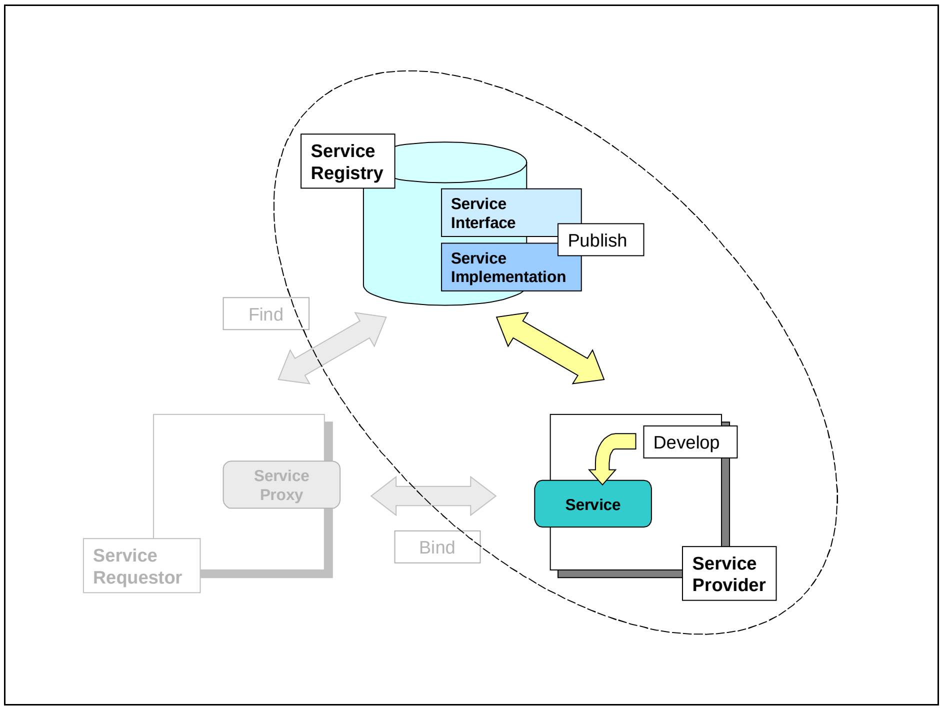

Figure 1. Green field method

Build

1. *Develop the new Web service.*
   The first step in the development lifecycle is to design and implement the application that represents the Web service. This step includes the design and coding required to implement the service, and the testing to verify that all of its interfaces work correctly.

2. *Define a new service interface.*
   After the new Web service has been developed, the service interface definition can be generated from the implementation of the service. The service interface should not be generated until the Web service development is complete because the interface must match the exact implementation of the service.

Deploy

1. *Publish the service interface.*
   The service interface definition needs to be published before the service is deployed. The service interface is used by a service requestor to determine how to bind to the service.

2. *Deploy the Web service.*
   Deploy the runtime code for the service and any deployment meta data that is required to run the service. An example of deployment meta data would be the deployment descriptor that is required to deploy a SOAP service. Some Web Services will require a deployment environment which provides support for functions such as billing, auditing, logging and security. After a service has been deployed, it is ready to be used by a service requestor.

3. *Create the service implementation definition.*
   The service implementation definition should be created based on how and where the service was deployed. The service implementation definition can contain references to more than one version of the deployed Web service. This allows the service provider to implement different levels of service for service requestors.

4. *Publish the service implementation definition.*
   The service implementation definition contains the definition of the network-accessible endpoint or endpoints where the Web service can be invoked.

Run

Run the Web service.
The runtime environment for the Web service consists of the platform on which it was deployed to run. As an example, if the Web service is a servlet, then it runs in the context of a Web application server. If the Web service is a SOAP service, then it runs in the context of a SOAP server.

Top-Down

Service Interface	Exists
Web Service	New

Table 3. Top-down method

As shown in Table 3, using this method, a *new* Web service can be developed that conforms to an *existing* service interface. This type of service interface is usually part of an industry standard, which can be implemented by any number of service providers. As Figure 2 shows, the service provider must find the service interface, implement the interface contained in this definition, and then deploy the new Web service. The service interface can not be owned by the service provider.

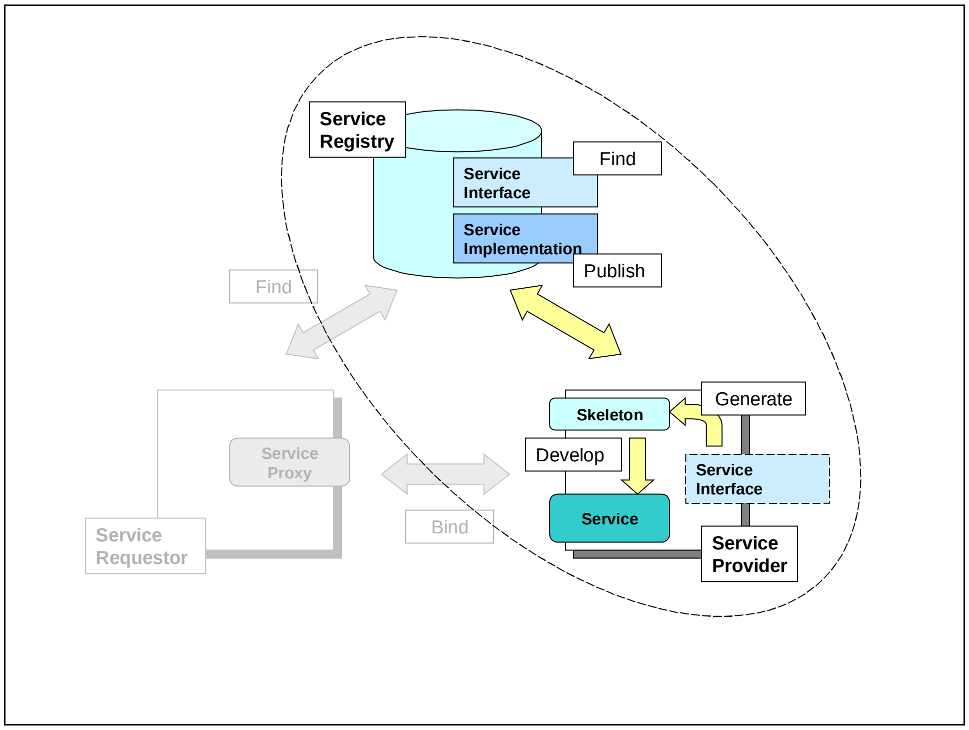

Figure 2.Top-down method

Build

1. *Find the service interface.*
   Locate the service interface that will be used to implement the Web service. The service interface is located by searching the service registry or an industry specification registry. The search can be completed by using keywords or taxonomy information.

2. *Generate the service implementation template.*
   Using the service interface definition, an implementation template of the Web service is generated. This will contain all of the methods and parameters that must be implemented by the Web service to be compliant with the service interface.

3. *Develop the new Web service.*
   Using the service implementation template created in the previous step, design and implement the application that represents the Web service. This step includes the design and coding required to implement the service, and the testing to verify that all of its interfaces work correctly.

Deploy

The deployment steps for the top-down method are the same as deployment steps 2 to 4 for the green field method. For the deploy phase, the only difference is that the service interface has already been published by another entity.

1. *Deploy the Web service.*
   Deploy the runtime code for the service, and any deployment meta-data that is required to run the service.

2. *Create the service implementation definition.*
   The service implementation definition should be created based on how and where the service was deployed.

3. *Publish the service implementation definition.*
   The service implementation definition contains the definition of the network-accessible endpoint or endpoints where the Web service can be invoked.

Run

Run the Web service.
The runtime environment for the Web service consists of the platform on which it was deployed to run.

Bottom-Up

Service Interface	New
Web Service	Exists

Table 4. Bottom-up method

As shown in Table 4, the bottom-up method is used to create a *new* service interface for an *existing* application. The application can be implemented as an Enterprise JavaBean™ (EJB), JavaBean, servlet, C++ or Java class file, or Component Object Model (COM) class. The service interface is derived from the application's application programming interface (API). Figure 3 shows the generation of the application.

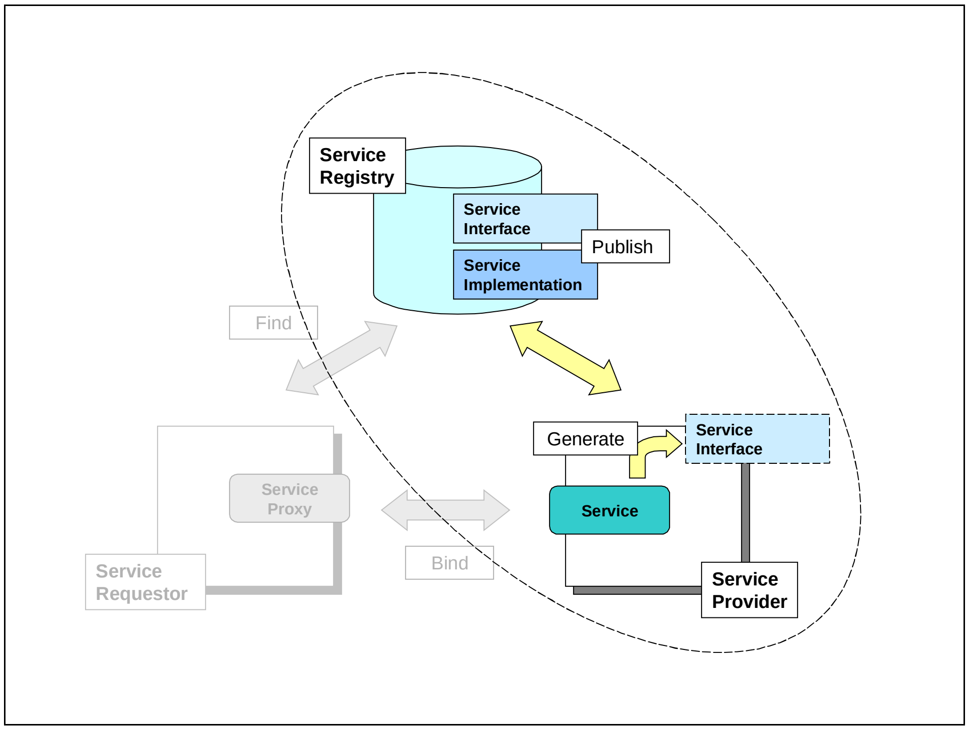

Figure 3. Bottom-up method

Build

Generate the service interface.
The service interface is generated from the implementation of the application that represents the Web service.

Deploy

1. *Deploy the Web service.*
   Deploy the runtime code for the service, and any deployment meta data that is required to run the service. After a service has been deployed, it is ready to be used by a service requestor.

2. *Create the service implementation definition.*
   The service implementation definition should be created based on how and where the service was deployed. The service implementation definition can contain references to more than one version of the deployed Web service. This allows the service provider to implement different levels of service for service requestors.

3. *Publish the service interface definition.*
   Before the service implementation can be published, the service interface definition must be published.

4. *Publish the service implementation definition.*
   The service implementation definition contains the definition of the network-accessible endpoint or endpoints where the Web service can be invoked. After the service implementation definition is published, a service requestor can find the service definition and use it to bind to the Web service.

Run

Run the Web service.
The runtime environment for the Web service consists of the platform on which it was deployed to run.

Meet-in-the-Middle

Service Interface	Exists
Web Service	Exists

Table 5. Meet-in-the-middle method

As shown in Table 5, the meet-in-the-middle method for developing a Web service is used when a service interface already *exists* and the application that will be used for the Web service already *exists*.

The primary task for this method of developing a Web service is to map the existing application interfaces to those defined in the service interface definition. As shown in Figure 4, this can be done by creating a wrapper for the application that uses the service interface definition, and contains an implementation that maps the service interface into the existing application interface.

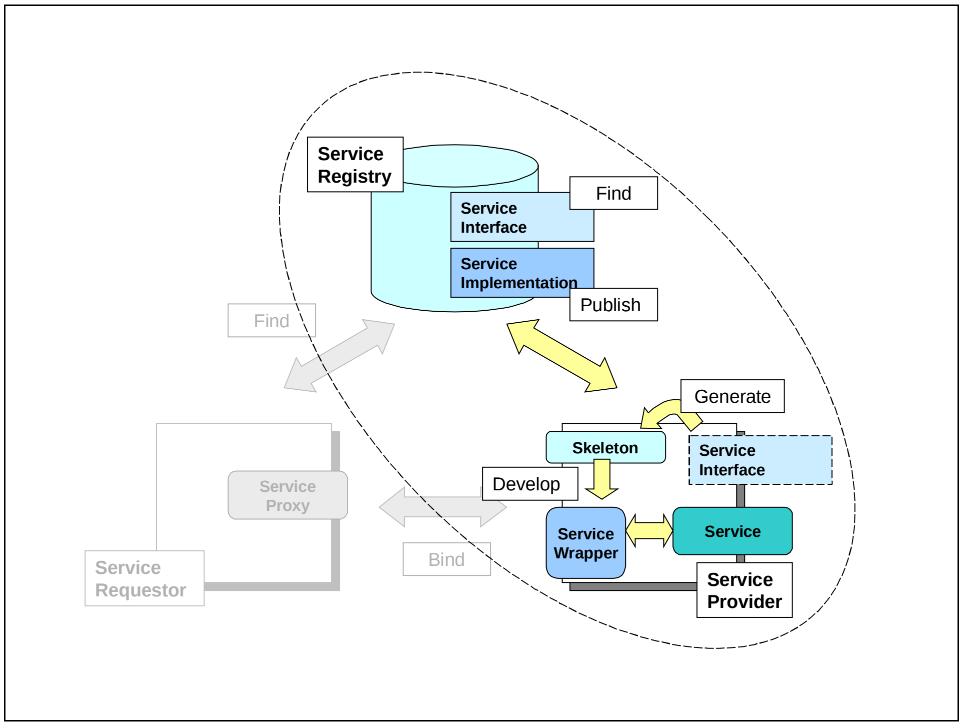

Figure 4. Meet-in-the-middle method

Build

The first two build steps are the same as those for the top-down method.

1. *Find the service interface.*
   Locate the service interface that will be used to implement the Web service.

2. *Generate the server implementation template.*
   Using the service interface definition, an implementation template of the Web service generated.

3. *Develop the service wrapper.*
   Using the service implementation template created in the previous step, design and implement the service wrapper which will map the service interface into the existing application interface.

Deploy

The deployment steps for the meet-in-the-middle method are the similar to those for the bottom-up method. The only difference is that the service interface is already published.

1. *Deploy the Web service.*
   Deploy the runtime code for the service, and any deployment meta-data that is required to run the service.

2. *Create the service implementation definition.*
   The service implementation definition should be created based on how and where the service was deployed.

3. *Publish the service implementation definition.*
   The service implementation definition contains the definition of the network-accessible endpoint or endpoints where the Web service can be invoked.

Run

Run the Web service.
The runtime environment for the Web service consists of the platform on which it was deployed to run.

Service Requestor

The service requestor progresses through the same lifecycle elements as the service provider, but the requestor performs different tasks during each phase. The build time tasks for the service requestor are dictated based on the method for binding to a Web service.

The service interface is used to generate a service proxy. The service proxy contains all of the code that is required to access and invoke a Web service. As an example, if the Web service is a SOAP service, the service proxy will contain all of the SOAP client code that is required to invoke a method on the SOAP service.

There are three methods for binding to a specific service. A *static binding* is used only at build time, whereas *dynamic binding* can be used either at build time or runtime. A static binding can not be used at runtime, because it requires all of the information needed to bind to a service at build time.

	Static Binding	**Dynamic Binding**
Build	Static binding	Build-time dynamic binding
Run	[not applicable]	Runtime dynamic binding

Table 6. Service requestor methods

Static Binding

A service requestor will use a static binding (Figure 5) when there is only one service implementation that will be used at runtime. The static binding is built at build time by locating the service implementation definition for the single Web service that will be used by the service requestor. The service implementation definition contains a reference to the service interface, which will be used to generate the service proxy code. The service proxy contains a complete implementation of the client application that can be used by the service requestor to invoke Web service operations.

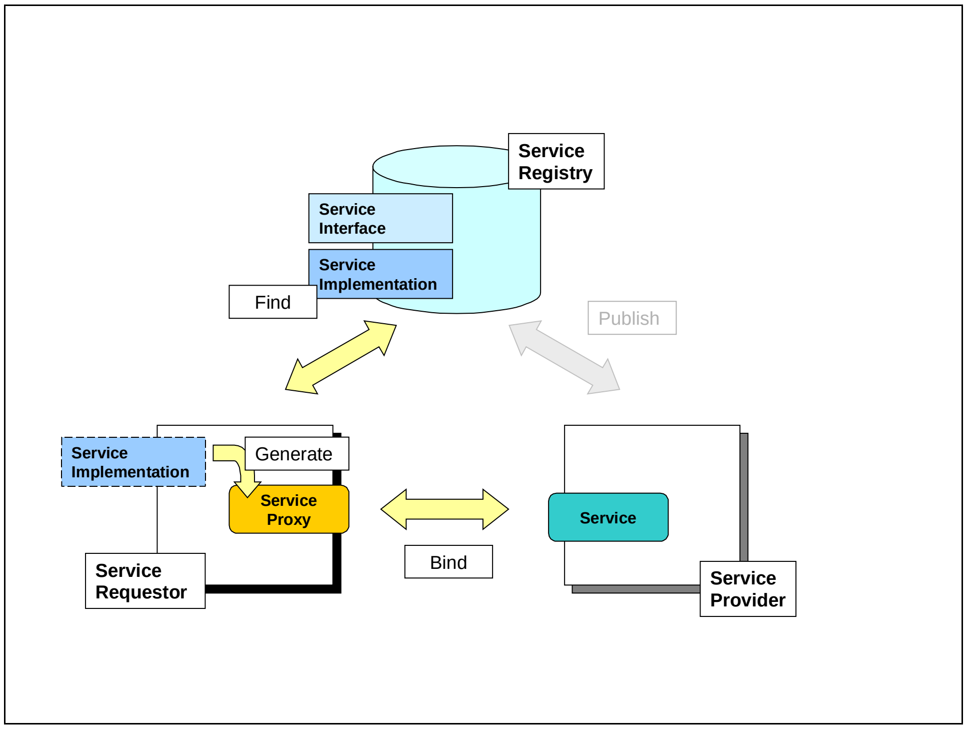

Figure 5. Static binding

Build

1. *Find the service implementation definition.*
   At build time, the service requestor must locate the service implementation definition for the Web service. The service implementation definition will contain both a reference to the service interface definition, and the location where the service can be accessed.

2. *Generate the service proxy.*
   The service interface definition and the service location information are used to generate the service proxy implementation. The service proxy implementation will conform to the service interface, and will always try to access the Web service at the same location.

3. *Test service proxy.*
   Before deploying the service proxy it should be tested to verify that it can interact with the specified Web service.

Deploy

Deploy service proxy.
After the service proxy has been tested to verify that it works correctly, it should be deployed with the client application in the client runtime environment.

Run

Invoke the Web service.
Run the requestor application which will use the service proxy to invoke the Web service.

Build-Time Dynamic Binding

This type of binding is used when a service requestor wants to use a specific type of Web service, but the implementation is not known until runtime or it can change at runtime. The type of service is defined in a service interface definition.

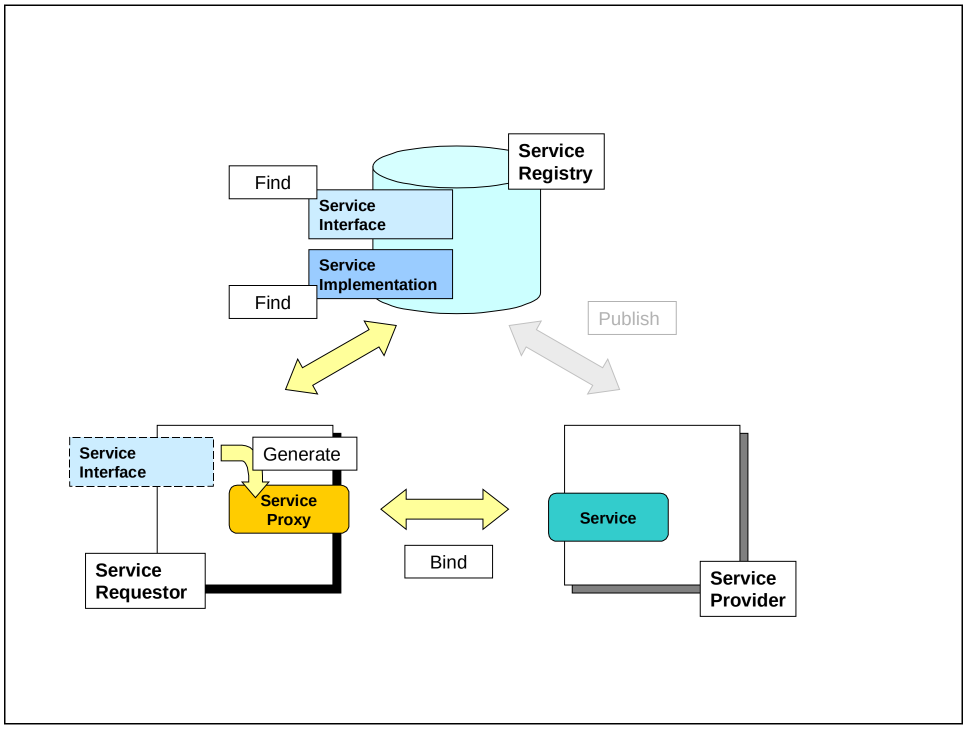

Figure 6. Build-time dynamic binding

Build

1. *Find the service interface definition.*
   The first step is to locate the service interface definition for the type of service that will be used by the service requestor. The service interface contains only the abstract definition of the Web service operations.

2. *Generate the generic service proxy.*
   Using the service interface definition, a generic service proxy can be generated. This service proxy can be used to access any implementation of the service interface. The only difference between this service proxy and the one generated for a static binding is that the static binding will contain knowledge of a specific service implementation. This means that the generic service proxy will contain code to locate a service implementation by searching a service registry.

3. *Test service proxy.*
   Before deploying the service proxy it should be tested to verify that it can interact with the specified type of Web service. This can be accomplished by finding an implementation of the service interface.

Deploy

Deploy service proxy.
After the service proxy has been tested to verify that it works correctly, it should be deployed within the runtime environment. This process can also include the deployment of the requestor application that will use the service proxy. The runtime environment that the service proxy is deployed in must have access to the service registry that will be searched for an implementation of the service interface.

Run

1. *Find the Service implementation definition.*
   Before the service proxy can invoke a service, an implementation of the service must be located in the service registry. The generated service proxy should contain all of the code that is required to search the service registry for an implementation of the service interface.

2. *Invoke the Web service.*
   After a service implementation has been found, the service proxy can be used to invoke the Web service.

Runtime Dynamic Binding

Runtime dynamic binding is similar to build-time dynamic binding. A service interface is used to generate a general service proxy interface that can be used to invoke any implementation of the service interface. This type of binding is different in that the service interface is found at runtime. After the service interface is found, the proxy code is generated, compiled, and then executed. This type of binding would typically be used with a user interface, because it is not possible for a machine-to-machine interaction to be truly dynamic.

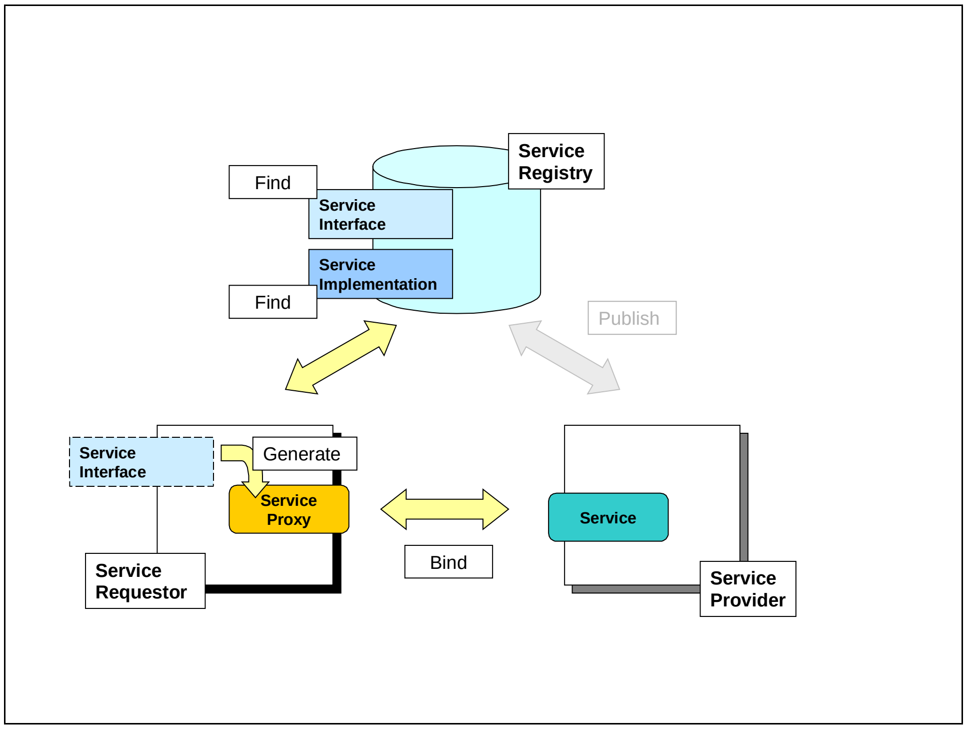

Figure 7. Dynamic runtime binding

Build

Build requestor application.
The service requestor application is built using a dynamic binding runtime interface. This interface is used to find a service implementation, and then retrieve the service interface associated with the service implementation.

Deploy

Deploy requestor application.
Deploy the requestor application so that it will run and use the Web service runtime environment.

Run

1. *Find the service implementation definition.*
   The service requestor application uses runtime environment to find a service implementation definition. There are different methods that can be used to locate a service implementation in a service registry. A service implementation can be found by first locating a business or type of business, and then determining the services offered by those businesses. The service implementation could also be located by searching for a classification of service, or by first locating a type of service (or service interface). If the service interface is the target of a search operation, then it is used to locate the implementations of service interface.

2. *Generate and deploy the service proxy.*
   Using the service interface associated with the service implementation, generate the service proxy code that will be used to invoke the service. After the code is generated, it is compiled and made available in the runtime environment.

3. *Invoke the Web service.*
   The generated service proxy code is used to invoke the Web service.

Web Service Tools

A set of service development tools can be used to assist with these scenarios. Development tools automate various aspects of Web service development simplifying design, deployment and integration. IBM provides a suite of development tools for Web Services, providing support for wrapping preexisting applications, generation of service descriptions, code generation from Web Services Description Language (WSDL) documents and more. The following components are logical functions that make up the tooling needed to support service development. They form functional parts that are planned to be integrated into the IBM XML and Web Services Development Environment.

UDDI Browser

This browser allows the developer to interactively browse the Universal Description, Discovery and Integration (UDDI) registry to find the services that can already be defined. This allows the developer to download an interface definition that would be the basis for the development of a conforming service.

UDDI Editor

The UDDI editor is used by the service developer to create different UDDI entries, including the *businessEntity*, *businessService* and *tModel* information needed to publish the service in a UDDI registry.

UDDI Publishing Tools

UDDI publishing tools take UDDI definitions created by the UDDI editor or generation tools and publish them to a UDDI registry. Because portions of these definitions can already exist in the registry, appropriate new or changed elements will be applied to the target UDDI repository. Examples of existing entities with new elements include a business entity to which we are adding additional services, and a binding template to which we are adding additional *tModel* references. Some elements can already exist, and will need to be resolved to existing universal unique identifiers (UUID) keys. The publishing tools must also support promotion of the UDDI information from a private test UDDI registry to one or more production registries.

WSDL Editor

This creates WSDL interfaces for publication. It is used by developers who are creating WSDL documents to describe Web Services from scratch.

WSDL Generator

The WSDL generator produces WSDL interface documents that describe interfaces implemented by existing applications. This tool can be used to automatically generate a WSDL document describing EJBs, JavaBeans, servlets, C++ or Java class files, Common Object Request Broker Architecture (CORBA), Interface Definition Language (IDL), COM class, and

MQSeries message definitions that have already been implemented. This tool also generates WSDL implementation documents, including support for non-IBM environments such as Microsoft .NET.

For SOAP services, the WSDL generator should also create the SOAP deployment descriptor which is required to deploy the service.

Service Proxy Generator

The service proxy generator produces client code from a WSDL interface document, and optionally a service implementation document. If only the service interface document is used, then a generic service proxy is generated. This type of proxy can be used to access any implementation of service interface. If both a service interface and a service implementation are used, then a service proxy is generated that will access only the specified service implementation. The service proxy contains the code that is specific to a binding within the service interface. For example, if the binding is a SOAP binding, then the service proxy will contain SOAP client code that is used to invoke the service.

Service Implementation Template Generator

The service implementation template generator can be used to create an implementation template used to implement a Web service. The implementation template is created using only the service interface definition. An example of a service implementation template is a Java interface. If binding in the service interface is a SOAP binding, then the deployment descriptor that is needed to deploy the SOAP service is also generated.

Developing Web Services Using Java

This section describes one instantiation of the abstract approach for developing Web Services defined in the previous section. It defines the build, deployment and runtime requirements of the service requestor and service provider as Java components.

Homogeneity across the service requestor and service provider component is not a requirement with respect to programming language or component model. Because this is a derivation of the Web Services development approach, the loosely coupled characteristics of Web Services are maintained such that a requestor can employ a non-Java-based Web Services development approach while maintaining interoperability with a peer Web service being hosted by a Java-based service provider.

Service Provider

Within this development approach, the service provider hosts Web Services within a Java runtime environment. The Java runtime environment consists of the following components:

- *Web application server*
  An HTTP server and servlet engine.

- *SOAP RPC router*
  Network endpoint that unmarshalls SOAP remote procedure call (RPC) messages and forwards them to a designated provider. The provider response is then marshalled and returned to the requestor. The router is conventionally a servlet managed within the context of the Web application server.

- *SOAP message router*
  Network endpoint that unmarshalls SOAP messages and forwards them to a designated provider. The router is conventionally a servlet managed within the context of the Web application server.

- *Plugable provider interface*
  Provides extensibility of the SOAP server, by facilitating the addition of user-defined providers. This support is required to host JavaBeans, servlets, EJBs and Java applications as Web Services. A pluggable provider will provide the necessary logic to locate, load and invoke the service implementation referenced in the SOAP request.

  If RPC-based, the provider will convert the service result to a SOAP envelope. This conversion is typically defined by an object model. For example, Apache SOAP 2.1 provides an `org.apache.SOAP.RPC.SOAPContext` object to encapsulate the envelope.

  The following are examples of pluggable providers that are used to interact with Java objects:

  - *Java RPC provider*
    Provides the bridge between the SOAP RPC router and the Java RPC service implementation being invoked. This provider is capable of loading the Java class specified in the SOAP message and invoking the appropriate operation, with the given parameters. The service response is then returned to the SOAP RPC router.

  - *Java message provider*
    Provides the bridge between SOAP message router and the Java messaging service being invoked. This provider is capable of loading the Java class specified in the SOAP message. The provider relays the message payload in the SOAP envelope to the target service.

- *EJB container*
  Hosting environment that manages EJB lifecycles. Required for EJB-based Web service implementations.

A Web service deployed in this environment will be implemented using one or more of the following methods:

1. Java class
2. Servlet
3. Bean Scripting Framework (BSF) compatible script
4. JavaBean
5. EJB
6. Java Native Interface (JNI) interface defined for native application components

Each of the following sections describes a method for developing a Web service using Java. When describing the steps within each phase of the lifecycle, some methods use the same steps. The repeated steps are described in detail only when they are listed for the first time.

Green Field

Build

1. *Develop the new Java Web service.*
   The first step in the development lifecycle is to design and implement the Java-based Web Services application. The service implementation can range from a simple Java class to an EJB.

2. *Define a new service interface.*

   The service interface is a WSDL document defining the type, messages and port type elements. After the development of the Web service, a service interface definition should be created. The service interface will describe the abstract operations supported by the service and the messages that will be communicated.

   For RPC interactions the messages represent input, input/output, and output parameters. This process can be manual or automated via WSDL utilities. Given a fully qualified class name for a Java class, bean, or servlet, a WSDL generation utility can use reflection to generate the service interface. The desired methods are selected prior to the generation of the service interface. Given an EJB deployment descriptor (that is, jar file), a WSDL generation utility can create the service interface that exposes the desired remote and home interface methods as service operations. IBM provides utilities which will generate the appropriate WSDL interface document for the given implementation types.

Deploy

1. *Publish service interface.*

   Publishing the service interface definition is a function supported by the registry component of the development approach. The registry can be a simple storage mechanism, without any supporting Web service data structures (for example, diskette or file system). However the registry can be more complex, and provide support for business registration, industrial categorization based on standard taxonomies, and maintaining technical service information. Some registries such as the UDDI registry will support a programmatic interface to publishing service interfaces. UDDI defines the UDDI programmer's API that incorporates a publish API, available at www.uddi.org. IBM provides tools that simplify publishing to UDDI registries via the UDDI4J implementation of the UDDI programmers API. Note that within the context of UDDI, a WSDL service interface document represents a *tModel*.

2. *Deploy Web service.*

   Deploying the Web service implies integration of the Web service and relevant meta-data with the Java runtime. The following characterizes the general deployment steps:

   i. *Generate SOAP deployment descriptor.*
      The SOAP deployment descriptor contains meta-data relevant to deploying services such as an object ID, provider type identification, and class identifier using the necessary naming system (that is, Java class name, Java Naming and Directory Interface [JNDI] environment naming context, and so on), and which class, instance method, or both that will be exposed as service operations.

      EJB service implementations will require the use of the EJB deployment descriptor to generate the SOAP deployment descriptor. Methods of the remote and home interfaces will be exposed as service operations. The SOAP deployment descriptor for EJBs must reference an EJB provider. The Apache SOAP implementation provides three EJB-compliant providers: stateless, stateful, and entity providers.

   ii. *Deploy SOAP service.*
      Deploy the service as a SOAP service. This can be accomplished using a server-side API provided by the SOAP server vendor. For example, in Apache SOAP the `ServiceManagerClient` class will deploy a service given a class name and deployment descriptor.

      EJB implementations of Web Services will require the use of an EJB provider class, which will unmarshall incoming SOAP request and forward the message to the appropriate EJB as an EJB client. The response data is then marshalled and returned to the SOAP client. The EJB is deployed within an EJB container.

3. *Create service implementation definition*

   The service implementation is a WSDL document defining the binding, service and port elements. The service implementation will import a service interface document. Generating the service implementation definition can be either a manual or automated process. This information will provide location information for the service and reference the service implementation definition. The SOAP address should reference the URL of the appropriate SOAP router. The binding should reference the object ID provided in the

SOAP deployment descriptor as the namespace for the operations. The binding is typed such that it is mapped to a particular port type defined by the service interface. IBM is planning to provide tools that will simplify the generation of service implementations.

4. *Publish service implementation definition.*

   Publishing the service implementation definition is a function supported by the registry component of the development approach.

Run

The Web service can have multiple runtimes or simply run within the context of a Web application server. For example, EJB service implementations will require the SOAP server to execute in a Web application service, while the EJB implementation itself can run in a separate runtime environment consisting of an EJB container.

Top-Down

Build

1. *Find service interface.*

   Locating the service interface definition is a function supported by the registry component of the development approach.

2. *Generate service implementation template.*

   A service implementation template is typically generated through an automated process. The generated implementation template will be a Java class with empty methods defined, where the method signatures are mapped to the WSDL-defined operations and messages. Given a WSDL implementation, the deployment descriptor can be generated as well.

3. *Develop the new Web service.*

   Develop the implementation of the service by implementing the empty methods from the Java class generated in step 2. The service implementation can range from a simple Java class to EJB.

Deploy

1. *Deploy Web service.*
   Deploying the Web service implies integration of the Web service and relevant meta-data with the Java runtime. This process is the same as the one defined for the green field method.

2. *Create service implementation definition.*
   The service implementation is a WSDL document defining the binding, service and port elements.

3. *Publish service implementation definition.*
   Publishing the service implementation definition is a function supported by the registry component of the development approach.

Run

The Web service can have multiple runtimes or simply run within the context of a Web application server. For example, EJB service implementations will require the SOAP server to execute in a Web application service, while the EJB implementation itself can run in a separate runtime environment consisting of an EJB container.

Bottom-Up

Build

1. *Generate service interface.*

   The service interface is a WSDL document defining the type, messages and port type elements. After the development of the Web service, a service interface definition should be created. The service interface describes the abstract operations supported by the service and the messages that will be communicated. For RPC interactions, the messages represent input, input/output, and output parameters. This process can be manual or automated using WSDL utilities.

   Given a fully qualified class name for a Java class, bean or servlet, a WSDL generation utility can use reflection to generate the service interface. The desired methods are selected prior to the generation of the service interface.

   Given an EJB deployment descriptor (that is, jar file) a WSDL generation utility can create the service interface that exposes the desired remote and home interface methods as service operations. IBM provides utilities that will generate the appropriate WSDL interface document for the given implementation types.

Deploy

1. *Deploy Web service.*

   Deploying the Web service implies integration of the Web service and relevant meta-data with the Java runtime.

2. *Create service implementation definition.*

   The service implementation is a WSDL document defining the binding, service and port elements.

3. *Publish service interface.*
   Publishing the service interface definition is a function supported by the registry component of the development approach.

4. *Publish service implementation definition.*
   Publishing the service implementation definition is a function supported by the registry component of the development approach.

Run

The Web service can have multiple runtimes or simply run within the context of a Web application server. For example, EJB service implementations will require the SOAP server to execute in a Web application service, while the EJB implementation itself can run in a separate runtime environment consisting of an EJB container.

Meet-In-The-Middle

Build

1. *Find service interface.*

   Locating the service interface definition is a function supported by the registry component of the development approach.

2. *Generate service implementation template.*
   A service implementation template is typically generated through an automated process. The generated implementation template will be a Java class with empty methods defined, where the method signatures are mapped to the WSDL-defined operations and messages.

3. *Develop the service wrapper.*
   The service wrapper represents the mapping code for the preexisting Java code base. The empty methods from the Java class generated in the previous section should be the integration points for the preexisting application. For EJB service implementations, the service wrapper should use the home and remote interfaces supported by the enterprise bean implementation.

Deploy

1. *Deploy Web service.*

   Deploying the Web service implies integration of the Web service and relevant meta-data with the Java runtime.

2. *Create service implementation definition.*

   The service implementation definition is a WSDL document defining the binding, service and port elements.

3. *Publish service implementation definition.*

   Publishing the service implementation definition is a function supported by the registry component of the development approach.

Run

The Web service can in fact multiple runtimes or simply run within the context of a Web application server. For example EJB service implementations will require the SOAP server to execute in a Web application service, while the EJB implementation itself can run in a separate runtime environment consisting of an EJB container.

Service Requestor

Within this development approach, the service requestor is hosted within a Java runtime environment. The Java runtime environment for the service requestor assumes only a Java virtual machine. Because WSDL defines bindings for SOAP, HTTP GET/POST and MIME extensions, the service requestor can access services from a browser or application runtime. A Web service requestor deployed in this environment represents one or more of the following:

1. Java class
2. Servlet
3. JavaServer Pages (JSP)
4. JavaBean
5. EJB
6. JNI interface defined for native application components
7. Java Applet

Static Binding

Build

1. *Find service implementation definition.*

   Locating the service implementation definition is a function supported by the registry component of the development approach.

2. *Generate service proxy.*

   A service proxy is typically generated through an automated process. The generated proxy will be a Java class where the implemented methods have signatures that are correlated with the service interface (imported by service implementation). The body of the methods leverage the service implementations binding and address information to build a SOAP message. The SOAP message can be represented by an object model. Apache SOAP 2.1 provides the `org.apache.SOAP.RPC.Call` class to encapsulate the SOAP request message.

3. *Build requestor application.*

   Develop the requestor application such that it uses the service proxy methods to invoke the Web service.

4. *Test service proxy.*

   Before deploying the service proxy it should be tested to verify that it can interact with the specified Web service.

Deploy

Deploy service proxy.

Deploy the service proxy within a Java Virtual Machine (JVM) along with the requestor Java application. A SOAP client is required to process the SOAP messages.

Run

Invoke Web service.

The requestor runs within the context of a JVM.

Build-Time Dynamic Binding

Build

1. *Find service interface definition.*

   Locating the service interface definition is a function supported by the registry component of the development approach.

2. *Generate generic service proxy.*

   A service proxy is typically generated through an automated process. The generated proxy will be a Java class where the implemented methods have signatures that are correlated with the service interface (imported by service implementation).

3. *Build requestor application.*

   Develop the requestor application such that it uses the service proxy methods to invoke the Web service.

4. *Test service proxy.*

   Before deploying the service proxy it should be tested to verify that it can interact with the specified Web service.

Deploy

Deploy service proxy.

Deploy the service proxy within a JVM along with the requestor Java application.

Run

1. *Find service implementation definition.*

   Locating the service implementation definition is a function supported by the registry component of the development approach.

2. *Invoke Web service.*

   Now that the network address of the desired service implementation has been resolved, the service can be invoked.

Runtime Dynamic Binding

Build

Build requestor application.

Build the requestor Java application with introspection functionality to facilitate leveraging a mutable API.

Deploy

Deploy requestor application.

Deploy the requestor application within a JVM.

Run

1. *Find service interface definition.*

   Locating the service interface definition is a function supported by the registry component of the development approach.

2. *Find service implementation definition.*

   Locating the service implementation definition is a function supported by the registry component of the development approach.

3. *Generate and deploy service proxy.*

   A service proxy is typically generated through an automated process. The generated proxy will be a Java class where the implemented methods have signatures which are correlated with the service interface (imported by service implementation). The service proxy should be compiled and integrated within the requestor application using introspection and reflection, and the following method: `Class.getConstructer(Class[]).newInstance(Object[]).`

4. *Invoke Web service.*

   Now that the network address of the desired service implementation has been resolved and the service proxy can be programmatically accessed, the requestor can invoke the service.

Related Information

Web Sites

AXIS	http://xml.apache.org/axis
DISCO	http://msdn.microsoft.com/xml/general/disco.asp
EbXML	http://www.ebxml.org/
JAVA	http://java.sun.com/
OAG	http://www.openapplications.org/
SOAP	http://www.w3.org/TR/SOAP/
UDDI	http://www.uddi.org/
WSDL	http://www.uddi.org/submissions.html
XMLP	http://www.w3.org/2000/xp/
XML Schema Part 1	http://www.w3.org/TR/xmlschema-1/
XML Schema Part 2	http://www.w3.org/TR/xmlschema-2/

Other Papers

Web Services Flow Language (WSFL 1.0)

Web Services Conceptual Architecture (WSCA 1.0)

IBM Web Services Roadmap

Web Services and Business Process Management Technology