Web Services support in Borland® Enterprise Server 6.0

A Borland White Paper

By Vishy Kasar

February 2004

Borland®

Contents

Introduction **4**

The integration challenge **4**

Prior solutions: RPC and MOM **5**

Service-Oriented Architectures (SOA) **6**

Protocol *7*

Service contract *7*

Discovery mechanism *7*

SOA using Web Services **8**

SOAP: Message protocol *8*

WSDL: Service contract *9*

UDDI: Discovery mechanism *11*
businessEntity 11
businessService 11
bindingTemplate 11
tModels 12

Java™ specifications for Web Services **12**

SAAJ *12*

JAX-RPC *13*

JAXR *14*

Apache Axis **14**

Runtime *15*

Tools *15*

Borland Enterprise Server enhancements *15*

Borland Enterprise Server Axis tooling support *18*
Exposing the stateless session bean as an EJB 18
Accessing WSDL of all services inside a WAR 18
Viewing the WSDD 18

Editing the WSDD 18
Accessing the TCP Monitor 18

jUDDI 19

Borland Enterprise Server jUDDI tooling support 19

Usage scenario 20

Summary 21

Introduction

Web Services is a next-generation integration technology that helps glue together the numerous distributed and heterogeneous software applications that exist in a modern IT environment. Engineered to facilitate application integration using Web Services, Borland® Enterprise Server provides built-in functionality to help software teams both develop new Web Services-based applications and expose existing applications as Web Services.

This white paper describes the Web Services-related technology stack and how it can benefit IT organizations. It also identifies the types of problems that are best resolved using Web Services and details how Borland Enterprise Server can assist software teams in building, deploying, managing, and monitoring Web Services environments.

The integration challenge

Integrating software applications is a top priority for IT directors and CIOs in today's global enterprises. They recognize that well-integrated applications greatly enhance the productivity of employees. For example, a support representative can better assist the customer if he knows what problems he has experienced in the past, what products he has purchased, and where more-detailed information about those products can be found. However, all this information is probably stored in different software applications. For the support representative to jump from one application to another would be inefficient, especially with a customer waiting on the phone. Ideally, all this information would be easily accessible on a single computer screen.

Integrating the various applications would be fairly easy if they were all from same vendor or running on the same platform. Unfortunately, this is rarely the case. Today's organizations are faced with the problem of integrating applications from different vendors running on different platforms. Mergers and acquisitions result in multiple disparate technologies and applications in use within a single enterprise. Even when a new application is purchased, it is impractical

to throw away the existing application simply because it does not integrate readily with the new application.

The integration challenge becomes even more complex when you consider the supply-chain activities of a retailer such as Wal-Mart, which has thousands of partners and suppliers. The inventory and ordering system at Wal-Mart must interface with thousands of supplier systems. Each of those suppliers works with hundreds of retailers. It is not likely that all involved systems use similar technologies or run on the same platform.

Web Services were created to address this challenge: the integration of heterogeneous technologies and applications. Web Services make it possible to glue together these disparate technologies within the enterprise or across different companies. In a sense, Web Services enable the creation of a "grid" of interoperating technologies and applications, which facilitate the "frictionless" modern commerce environment.

Prior solutions: RPC and MOM

The "Integration problem" described above is not new. There are existing solutions that attempt to resolve these problems. Those solutions can be broadly classified as Remote Procedure Call (RPC)-style middleware and Message-Oriented Middleware (MOM).

RPC-style middleware typically packs the client request and parameters in binary packets and sends them to the server. The server then unpacks that binary data, invokes the required server object, and replies with a binary packet that contains the response. Middleware vendors typically provide tools to generate client stubs and server skeletons that pack and unpack the request and responses. Examples of RPC-style middleware include: Microsoft® DCOM, Object Management Group® (OMG®) CORBA, ® and Java™ Remote Method Invocation (RMI). Borland Enterprise Server 6.0 supports CORBA-style RPC.

Although RPC-style middleware is an excellent platform for deploying applications, it does not serve as a good integration technology. The binary protocols used by various middleware frameworks are different, and bridging between them is not easy. These middleware

frameworks also require a thick client library, which might make them difficult to use if the client is a (thin) Web browser. RPC-style middleware also promotes tight coupling between the server and client, whereas loose coupling might be more appropriate when connecting disparate applications from different vendors. Furthermore, some middleware frameworks (like Microsoft DCOM) run on only a single platform.

MOM systems send asynchronous messages between peers. The peers are loosely coupled, and there is no standard MOM protocol on which all vendors agree. Examples of MOM systems are TIBCO Rendezvous,™ SonicMQ,® Java™ Message Service (JMS) from multiple vendors, and Microsoft® Message Queuing (MSMQ). Borland Enterprise Server 6.0 bundles the TIBCO Enterprise™ for JMS as the messaging service.

The loosely coupled, asynchronous nature of MOM is exactly why it has been used as an integration technology. However, different protocols used by MOM vendors today make it difficult to use as such. Another consideration is that MOM-based solutions can be prohibitively expensive.

Web Services go beyond these prior technologies by providing a text-based message protocol that can be transported using the ubiquitous Internet. More important, all major vendors support Web Services standards. The next sections describe service-oriented architectures (SOA) and how Web Services can help fill in the various pieces of the integration puzzle.

Service-Oriented Architectures (SOA)

Although the term SOA has recently become popular, the SOA concept itself is not new. The term SOA defines a pattern where a server publishes its service contract (interface) to a broker in order to provide clients the ability to find that service. The following sections describe the fundamental components of SOA.

Protocol

The client and server must agree on the protocol used for communication. There are two types of protocols to consider:

- The **message protocol** defines the structure of the message and how to encode it on the wire.
- The **transport protocol** defines the mechanism used to send the message.

Service contract

A service contract clearly spells out the input messages, output messages, operations, and exceptions.

Discovery mechanism

SOA also defines a way for a service consumer to discover a service provider. This may not be necessary if consumers and providers are already familiar with one another. However, this discovery mechanism is useful when the consumer has no prior knowledge of a provider. Categorizing the data and the metadata about services in a well-defined broker is also important.

The actual components used in different service-oriented architectures obviously will be different. The following sections articulate how SOA is implemented with Web Services.

Service-Oriented Architecture using Web Services

The following diagram illustrates SOA components utilized by Web Services.

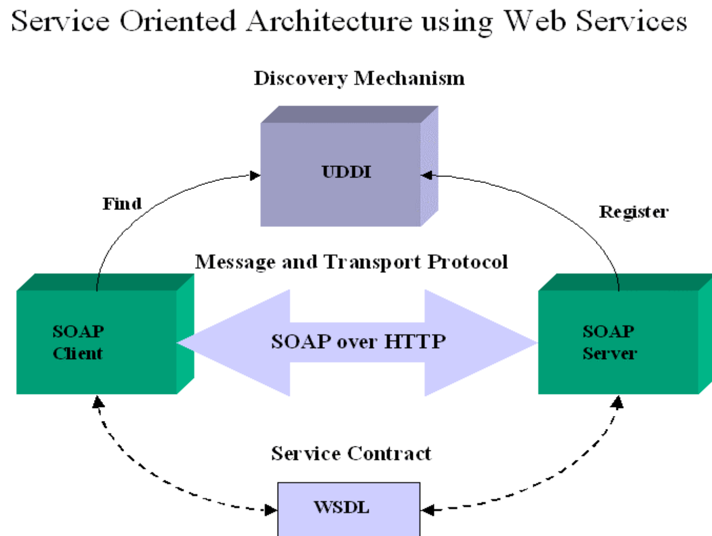

Figure 1: *Service-Oriented Architecture using Web Services*

SOAP: Message protocol

SOAP stands for *Simple Object Access Protocol.*

SOAP is an XML-based message protocol. It adds structure on top of XML to make it suitable for object access. It introduces the notion of an Envelope that includes Headers and Body. SOAP supports two mechanism styles that structure messages:

- The **RPC style** structures messages corresponding to a remote procedure. For example, the payload is structured to contain the remote method name followed by parameter name, type, and value.
- The **Document style** assumes that a pure document message is sent, with no notion of RPC. A good example of this is a purchase order document.

It is also possible to send attachments such as binary files using SOAP messages. The W3C® Note "SOAP Messages with Attachments" addresses the issue.

SOAP is a stateless message-level protocol; i.e., it does not maintain state between consecutive messages. SOAP can be communicated over any transport protocol, the most popular being HTTP. The following diagram depicts a SOAP packet sent over HTTP.

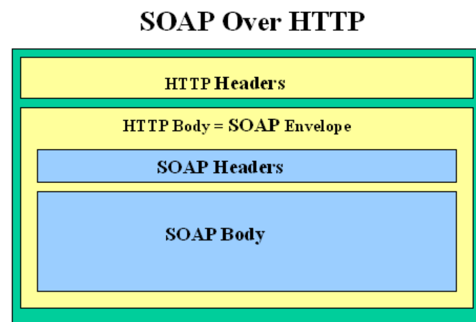

Figure 2: *A SOAP packet sent over HTTP*

Borland Enterprise Server 6.0 supports SOAP 1.1.

Why should you use SOAP?

Using SOAP, as a message-level protocol, opens up services to third-party applications running on any platform. Because no specific type of client is assumed, this creates a high degree of design flexibility and client decoupling.

WSDL: Service contract

WSDL stands for Web Services Description Language. It describes the "what," "how," and "where" of Web Services.

The “**what**” of WSDL deals with the **interface** of a Web Service. The interface describes the supported operations and the input/output parameters accepted by them. The interface essentially tells the service consumer what services are available. WSDL defines individual services as “**port types**.”

The “**how**” of WSDL deals with how a client is supposed to communicate with the server. It indicates what kind of transport is used to send messages, what style of message is to be sent (RPC or document), and how the messages are encoded on the wire. WSDL defines this information in “**binding**.”

The “**where**” of WSDL describes where the service resides. For SOAP over HTTP implementations, this information could be provided as the service URL. WSDL defines service location information in “**port**.”

It is possible to have multiple bindings for a given port type. These bindings may have different qualities of service and may be consumed by different types of clients. For example, an intranet client could use IIOP as the protocol, while an Internet client would use HTTP.

The biggest difference between WSDL and a traditional service contract such as IDL is that IDL provides only the “what.” With IDL, the “where” must be derived from another source, and the “how” is hard-coded based on the middleware. For example, if the middleware is CORBA, then CDR encoding will be used over an IIOP transport.

Borland Enterprise Server 6.0 supports WSDL 1.1

Why should you use WSDL?

Exposing services with WSDL lets your trading partners easily connect to them. It provides the what, how, and where information about an exposed service, effectively enabling different kinds of consumers to connect without the need for integration activity on the provider side.

UDDI: Discovery mechanism

UDDI stands for *Universal Description, Discovery, and Integration*.

For Web Services to be meaningful, it is important to provide information about them beyond the technical specifications of the service itself. Central to UDDI's purpose is the representation of data and metadata about Web Services. Data is worthless if it is lost within a mass of other data and cannot be distinguished or discovered. Based on this principle, UDDI allows the definition of any number of classification schemes (taxonomies), allows such classification schemes to be used on every component in the information model, and allows users to issue precise queries based on these schemes.

The following paragraphs describe the major components of UDDI information model.

businessEntity

A businessEntity is a top-level data structure used to represent businesses and service providers. It contains descriptive information about the business or provider and the services it offers. This would include information such as names and descriptions in multiple languages, contact information, and classification information.

businessService

A businessService structure represents a logical grouping of Web Services. It is contained within a businessEntity structure. Each businessService contains descriptive (nontechnical) information outlining the purpose of the individual Web Service found within it.

bindingTemplate

Each bindingTemplate structure represents an individual Web Service. A bindingTemplate provides the technical information needed by applications to bind and interact with the Web Service being described. It must contain either the access point for a given service or an indirection mechanism that will lead one to the access point. The bindingTemplate is a "child" of businessService.

tModels

Technical Models, or tModels, represent a shareable resource such as a WSDL specification. These resources can be shared by multiple businesses. For example, all the car manufacturers could agree on a standard specification for tires, create a tModel for that specification, and refer to that tModel from each of their bindingTemplates.

Borland Enterprise Server 6.0 supports UDDI 2.0.

Why should you use UDDI?

Advertising services in a UDDI server makes it easy for partners to find both business and technical information about those services. UDDI clients are able to discover and reuse services instead of building them from scratch.

Java™ specifications for Web Services

The discussion so far has dealt with Web Services concepts in a language-neutral fashion. The following sections discuss specifications that have emerged to make Web Services development easier using the Java programming language.

SAAJ

SOAP with Attachments API for Java (SAAJ) deals with representing SOAP packets in Java. SOAP itself is defined in a language-neutral fashion. However, a Java programmer interested in SOAP packets would need a corresponding Java representation. That is exactly what SAAJ specifies. As the name suggests, SAAJ not only covers SOAP, but SOAP with attachments as well. It defines Java objects such as SOAPMessage and provides APIs such as getSOAPBody on that object.

Borland Enterprise Server 6.0 supports SAAJ 1.1.

Why should you care about SAAJ?

The typical Web Services developer does not have to care about SAAJ for the most part. Borland Enterprise Server takes care of binding SOAPMessages to your business services automatically. However, if you need to write SOAP message handlers, then obviously SOAPMessages will be passed to your handler, and you need to be able to handle them. In such cases, you must use the SAAJ APIs.

JAX-RPC

Java API for XML-based RPC (JAX-RPC) provides standard mapping for XML/WSDL to and from Java. This allows a Java programmer who is completely unaware of XML, SOAP, and WSDL to develop Web Services in Java. In addition to defining mapping between the standard types, JAX-RPC also defines a framework for type expansion so that it is possible to define and plug-in custom serializers and deserializers that map complex Java types to XML and vice versa. It also introduces the notion of pluggable, chained SOAP handlers to provide common functionality such as security.

Note: Contrary to what the name suggests, JAX-RPC supports both RPC-style and document-style messages.

Borland Enterprise Server 6.0 supports JAX-RPC 1.0.

Why should you care about JAX-RPC?

When choosing a Java Web Services solution, you must ensure that it adheres to JAX-RPC. That will make sure that the type serializers, type deserializers, and SOAP handlers that are part of the application will continue to work on another JAX-RPC-compliant Web Services solution. Also, JAX-RPC Web Services are much more likely to interoperate with non-Java Web Services solutions.

JAXR

Java API for XML Registries (JAXR) provides a uniform and standard Java API for accessing different types of XML registries, including UDDI and ebXML. More such registries could be defined in the future. JAXR provides an API that enables clients to interact with XML registries and an SPI that enables providers to plug in their registry.

Borland Enterprise Server 6.0 supports JAXR 2.0.

Why should you care about JAXR?

JAXR API insulates application code from the underlying registry mechanism. If you are writing a JAXR-based client to browse or populate a registry, your code does not have to change if the registry changes, for example, from UDDI to ebXML. If you are developing a JAXR-based registry server, you increase your server's appeal by making it readily accessible.

The above discussion covered Web Services standards and the Java Language Specifications that deal with them. The next sections provide details about how Borland Enterprise Server implements these standards. The implementation of SOAP, WSDL, SAAJ, and JAX-RPC is provided through the Axis toolkit, which is bundled with Borland Enterprise Server. The implementation of UDDI and JAXR is provided through the jUDDI toolkit bundled with Borland Enterprise Server.

Apache Axis

Axis is a SOAP toolkit for Java from Apache (http://ws.apache.org/axis/) that has been integrated with many application servers. Axis provides a runtime component and several tools.

Runtime

The Axis runtime engine consumes SOAP messages that come over the wire. It invokes the appropriate handlers to assist with the flow of messages. It converts the XML body fragment to Java. It provides a proprietary mechanism to plug in Java handlers that can be called in response to a particular SOAP message. This proprietary mechanism is called the Web Services Deployment Descriptor (WSDD).

The Axis engine itself is a servlet that can be deployed easily inside a WAR into any servlet container. The servlet container strips the HTTP headers and passes the SOAP envelope to the engine.

Tools

Axis provides development tools (wsdl2java and java2wsdl) for converting WSDL to and from Java. It also provides a useful monitoring tool called TCP Monitor that lets you watch the HTTP traffic coming over the wire.

Why did Borland choose Axis?

The Axis toolkit adheres to SOAP 1.1, SAAJ 1.1, JAX-RPC 1.0, and WSDL 1.1. It has very good community support behind it, ensuring continuous evolution with the standards. Axis has a pluggable architecture that makes it easy to integrate it in to application servers. Instead of building yet another SOAP toolkit, Borland integrated the Axis toolkit into Borland Enterprise Server 6.0 and then documented the enhancements for the larger Axis community. “Pluggable Provider Architecture” is a good example of the contribution Borland has made to the Axis project.

Borland Enterprise Server enhancements

Borland Enterprise Server allows you to expose existing stateless Enterprise JavaBeans™ (EJB™) or Borland® VisiBroker® (CORBA) objects as Web Services. In order to make this possible, Borland Enterprise Server augments the Axis EJB provider architecture and adds a custom VisiBroker provider.

The EJB provider makes it possible for a client to access an existing stateless EJB using SOAP. It is not necessary to modify the EJB to make this happen. The following are the steps involved in exposing EJBs as Web Services.

1. Generate EJB client stubs
2. Generate a Web Services deployment descriptor (WSDD)
3. Bundle the stubs, WSDD, and Axis libraries in to a WAR
4. Deploy the WAR and EJB jar either independently or in one EAR

These steps are done automatically when using the Borland JBuilder Web Services Designer. The VisiBroker provider enables a Web Services client to make an invocation on a CORBA object without actually modifying that CORBA object. It is important to note that both **C++ and Java CORBA objects** can be exposed as Web Services. The following are the steps involved in exposing VisiBroker objects as Web Services.

1. Generate client stubs using the IDL
2. Generate a Web Services deployment descriptor (WSDD)
3. Bundle the stubs, WSDD, and AXIS libraries in to a WAR
4. Deploy the WAR
5. Run the VisiBroker server hosting the object being exposed

The following diagram shows how EJB and CORBA objects are exposed as Web Services in Borland Enterprise Server.

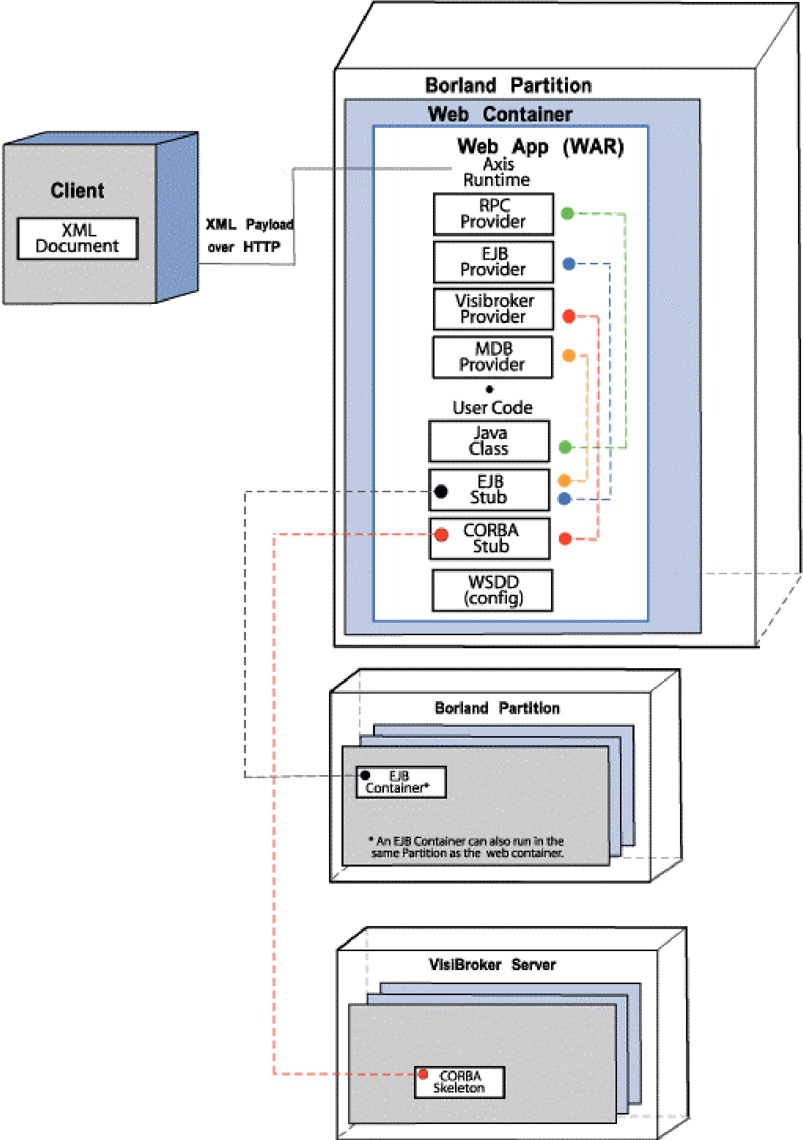

Figure 3: *How EJB and CORBA objects are exposed as Web Services in Borland Enterprise Server*

Borland Enterprise Server Axis tooling support

The Borland Enterprise Server Console and Deployment Descriptor Editor both support Axis-based Web Services.

Exposing the stateless session bean as an EJB

A user can select a stateless session bean from a deployed EJB jar in the Console and export it as a Web Service. After providing basic information, the stateless EJB is exposed as a Web Service without having to write a single line of code.

Accessing WSDL of all services inside a WAR

In the Borland Enterprise Server Console, under the general tab URLs, a user can see the URLs for the WSDL of all the Web Services in a WAR. These URLs end in "?wsdl". Clicking on any of those URLs brings up a browser that shows the WSDL that is dynamically generated by the server.

Viewing the WSDD

A user can select a Web Services-enabled WAR using the Borland Enterprise Server Console and see a special tab named "Axis Properties" on the right. The user is also able to drill down into various components that make up the WSDD.

Editing the WSDD

A user can select a Web Services-enabled WAR using the Borland Enterprise Server DDEditor and see a special tab named "Web Services" under the XML tab on the right. The user is also able to modify WSDD here and add a new service or change an existing service.

Accessing the TCP Monitor

The Axis TCP monitor tool can be accessed from the Borland Enterprise Server Console Tools menu. It is also available as the *tcpmonitor* executable under the Borland Enterprise Server install bin directory.

jUDDI

jUDDI is an open-source implementation of the UDDI 2.0-compliant server in Java. It is an Apache incubator project within the Apache Web Services Project. jUDDI can be used with any relational database that supports ANSI SQL. It can be deployed on any Servlet 2.3-compliant Web container. It is also possible to integrate jUDDI with an existing authentication system.

Borland Enterprise Server 6.0 has the capability to create the database required by jUDDI dynamically. jUDDI is provided as an EAR in the Borland Enterprise Server 6.0 repository. A user can deploy and manage jUDDI just like any other archive within Borland Enterprise Server 6.0.

Why did Borland choose jUDDI?

jUDDI is one of the most UDDI 2.0-compliant servers available. It offers a JDBC-compliant backing store and enables users to take advantage of any JDBC-compliant backing store already in use. Borland is an active contributor to the jUDDI project.

Borland Enterprise Server jUDDI tooling support

The Borland Enterprise Server 6.0 Console provides a Web Services Explorer that enables users to read and write to the UDDI registry in a user-friendly way. It also provides access to jUDDI proprietary extensions to the UDDI specification. These administrative extensions make it simple to add new "publishers," which can then be used to register and manage new Web Services. The Web Services Explorer is accessed from the tools menu of the Borland Enterprise Server Console.

Usage scenario

Imagine you are responsible for developing and maintaining an ordering system for the toy manufacturer SuperToys. SuperToys sells toys to retailers and accepts purchase orders by phone, fax, and the SuperToys Web site. Recently, however, several SuperToys retailers have requested direct access to the SuperToys ordering system. This would enable the retailers to maintain their inventory levels automatically, without human intervention. These retailers utilize a range of technology platforms, from mainframes to laptops, and multiple operating systems. Your job is to make it possible for them to order the toys directly.

SuperToys has a three-tier architecture: the client tier, the J2EE™ middleware tier, and the database tier. The J2EE middleware used is Borland Enterprise Server 6.0. Because the clients are varied, loose coupling is required. Web Services seems the ideal solution to this integration problem. SuperToys has already implemented Java Entity Beans to mirror database entries and provided some stateless session beans so that the servlet layer can access the EJB layer. You can use those stateless session EJBs as a good entry point into the system.

Using the Borland Enterprise Server Console, right-click on the session beans that you want to expose and choose "Export EJB as a Web Service." Choose a name for WAR and select the methods you want to expose. Click "finish," and Borland Enterprise Server will generate a WAR and deploy it for you. If you now select the WAR using the Console, you will see some URLs ending with '?wsdl'. This is the URL for WSDL that you can pass to clients either directly or through the UDDI server. If your session bean uses simple types, that is all you have to do. In some cases, you might have to add known type mappings (e.g., BeanSerializer) to map the types used in EJBs to XML and vice versa. You can use the Borland Enterprise Server DDEditor to add the type mappings.

Occasionally, you might have to change the code slightly: if, for example, the EJB method parameter is a bean type, but the bean is missing an empty constructor; or your EJB methods take arguments that are neither simple types nor bean types. In such cases, you must write a few lines of code to surface your EJBs. Borland® JBuilder® X can be used to add such code.

With the WSDL provided, SuperToys retail customers can choose the tool of their choice to connect to the server software and place their orders directly.

Summary

As more enterprises take a Web Services-based approach to integrating their software applications, Borland continues to enhance its application lifecycle management solutions to help software teams both develop new Web Services-based applications and expose existing applications as Web Services. Borland Enterprise Server 6.0, released in January 2004, provides built-in support for the following Web Services specifications:

- SOAP 1.1
- WSDL 1.1
- UDDI 2.0
- SAAJ 1.1
- JAX-RPC 1.0
- JAXR 2.0

For more details about how Borland Enterprise Server can assist your software teams in building, deploying, managing, and monitoring Web Services environments, visit http://www.borland.com or contact your local Borland sales representative.

Made in Borland® Copyright © 2004 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Microsoft, Windows, and other Microsoft product names are trademarks or registered trademarks of Microsoft Corporation in the U.S. and other countries. All other marks are the property of their respective owners. Corporate Headquarters: 100 Enterprise Way, Scotts Valley, CA 95066-3249 • 831-431-1000 • www.borland.com • Offices in: Australia, Brazil, Canada, China, Czech Republic, Finland, France, Germany, Hong Kong, Hungary, India, Ireland, Italy, Japan, Korea, Mexico, the Netherlands, New Zealand, Russia, Singapore, Spain, Sweden, Taiwan, the United Kingdom, and the United States. 21606