# TABLE OF CONTENTS

Introduction ................................................................. 3

Web Services Primer ....................................................... 3

a. WSDL—Web Services Definition Language ............................. 4
   WSDL—Services and Ports Section .................................... 4
   WSDL—PortType and Binding Section ................................. 4
   WSDL—Message Section ............................................... 4
   WSDL—Types Section .................................................. 5

b. SOAP ................................................................. 5
   SOAP—Request Message .............................................. 5
   SOAP—Response Message ............................................ 6

c. Universal Description, Discovery, and Integration—UDDI ............... 6

Creating Web Services in PB10 .......................................... 6

Registering Web Services in EAS 5.x with the Web Services Toolkit ........ 7

Consuming Web Services in PowerBuilder Client Applications ............. 8
   Finding Web Services with the Web Services Proxy Wizards .......... 9
   Finding Web Services with the UDDIProxy PowerBuilder Extension .... 10
   Aliasing Proxy Names for XML and SOAP Methods .................... 10
   SOAP Connectivity using pbsoapclient100.pbxml ...................... 10
   PowerBuilder to XML Data Type Mapping ................................ 12

Consuming Web Services in PB10—JSP Client .......................... 12
   Generate Custom Tag using JSP Web Services Proxy Wizard ........... 12
   Write JSP Page ....................................................... 12

Summary ........................................................................ 13

Web Services—A Natural Extension of PowerBuilder .......................... 13
INTRODUCTION

Distributed computing is not a new idea. Proprietary implementations and more widely adopted methods like DCOM and Object Request Brokers (ORBs) based on the CORBA specification represent earlier forays into splitting assigned tasks across multiple machines. Today’s functional approach to distributed computing is a Service-Oriented Architecture (SOA) based on Web Services. Web Services is a soup-to-nuts distributed computing architecture that arose as the Internet moved from childhood to adolescence and required a powerful, easily understood vehicle for exchanging information between Internet-based applications. Web Services information is easily interchanged between systems and can address a broad range of applications from simple needs, such as acquiring local weather information to be reformatted and presented as part of a portal’s window dressing, to highly complex analysis that requires aggregating information from multiple remote sources. Corporate versions of a service-oriented architecture expose multiple organizational assets in a highly interchangeable format that can be exchanged among back office systems or reformatted for human browsing.

With its extreme adoption of XML, Web Services burst on the scene as the right tool at the right time, utilizing the latest technology. This perfect fit made analysts rapturous to the point that Web Services was over-hyped and over-promised. Microsoft wholeheartedly embracing the technology with .NET only added to the frenzy. When the dust settled, and Web Services adoption moved at a healthier linear pace rather than the exponential saturation projected by the early predictions, true to human nature, there was a mild backlash as people recalibrated their expectations. There was nothing wrong with the technology—it had simply been over promised. Despite this early stumble, adoption continued at a sane, steady pace to the point that Web Services-based XML traffic is expected to comprise the majority of corporate XML traffic by the end of 2005, and XML traffic is projected to grow from 15% of the total Internet traffic in 2004 to 48% by 2008. Clearly, Web Services is coming into its prime.

Web Services is one of several possible front-end implementations of PowerBuilder applications. PowerBuilder 10, in conjunction with Sybase EAServer, fully supports creating, registering, and consuming Web Services and creating or augmenting a service-oriented architecture using PowerBuilder code. This whitepaper discusses implementing Web Services in PowerBuilder 10 and is intended for audiences interested in building Web Services into existing PowerBuilder applications, or creating new PowerBuilder applications to support Web Services. The first section covers Web Services technology and its individual components. The second section examines the requisite steps for mapping PowerBuilder functionality to Web Services.

WEB SERVICES PRIMER

Web Services are relatively simple, self-contained applications that perform targeted functions, from basic requests to complicated business processes. Web Services provide a standard method of interoperating between different software applications over the Internet or private networks. The technology is platform, language, and operating system independent. Unlike the traditional client browser/server relationship commonly used on the Internet, the Web Services model uses a peer-to-peer relationship.

The Web Services architecture has three primary components: description, transport, and discovery. Each of these components has an accompanying XML-based standard. Web Services are described with Web Services Description Language (WSDL), transported over HTTP and TCP/IP networks by Simple Object Access Protocol (SOAP), and discovered through Universal Description, Discovery and Integration (UDDI).

- **Web Services Definition Language (WSDL).** A common, self-referencing XML grammar to describe a service. WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.
- **Simple Object Access Protocol (SOAP).** Used as a standardized XML messaging system. It is a lightweight protocol for exchanging information in a decentralized, distributed environment.
- **Universal Description, Discovery, and Integration (UDDI).** UDDI is a simple find mechanism. UDDI employs registries or directories that contain information about Web Services and the businesses behind them.
A. WSDL—WEB SERVICES DEFINITION LANGUAGE

Web Services Definition Language (WSDL) is an unambiguous language for creating programatically readable descriptions of Web Services. It contains the messages, the types of information contained in the messages, and the nuts and bolts of the transmission mechanisms and protocols used to send the information. A single WSDL document can contain descriptions of multiple services.

A WSDL description is comprised of:

- **Service.** A collection of related Web service endpoints.
- **Port.** A single endpoint defined as a combination of a binding and a network address.
- **Port Type.** An abstract set of operations supported by one or more endpoints.
- **Binding.** A concrete protocol and data format specification for a particular port type.
- **Message.** An abstract, typed definition of the data being communicated.
- **Types.** Contains data type definitions in a type system such as XSD or schema.

**WSDL—SERVICES AND PORTS SECTION**

```xml
<service name="StockQuotes">
  <port name="StockQuotesSoap" binding="s0:StockQuotesSoap">
    <soap:address location="http://www.swanandmokashi.com/HomePage/WebServices/StockQuotes.asmx" />
  </port>
</service>
```

**WSDL—PORTTYPE AND BINDING SECTION**

```xml
<portType name="StockQuotesSoap">
  <operation name="GetStockQuotes">
    <input name="GetQuotes" message="s0:GetQuotesSoapIn" />
    <output name="GetQuotes" message="s0:GetQuotesSoapOut" />
  </operation>
</portType>

<binding name="StockQuotesSoap" type="s0:StockQuotesSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <operation name="GetStockQuotes">
    <input name="GetQuotes" >
      <soap:body use="literal" />
    </input>
    <output name="GetQuotes" >
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
```

**WSDL—MESSAGE SECTION**

This section details the supported messages including the types of information passed. The details of the information types are defined in the Types section of the WSDL.

```xml
<message name="GetQuotesSoapIn">
  <part name="parameters" element="s0:GetQuotes" />
</message>
```

```xml
<message name="GetQuotesSoapOut">
```
WSDL—TYPES SECTION

All of the type information supported by the service is defined in this schema section. This XSD contains the type definitions for information sent in the two example messages (GetQuotesSoapIn and GetQuotesSoapOut) defined in the message section.

```xml
<message>
    ...
</message>
```

B. SOAP

SOAP is a lightweight XML-based protocol for exchanging information in a decentralized, distributed environment, regardless of operating system or application language. SOAP can be delivered using a variety of transport protocols such as HTTP, FTP, and SMTP. The SOAP protocol contains three parts: an envelope structure that describes the contents of the message and how to process it, encoding rules for application-defined data types, and a representation of remote procedure calls and responses.

SOAP—REQUEST MESSAGE

```xml
mlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
mlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body>
        <m:GetQuotes xmlns:m="http://swanandmokashi.com">
            <m:QuoteTicker>SY</m:QuoteTicker>
        </m:GetQuotes>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
SOAP—RESPONSE MESSAGE
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetQuotesResponse xmlns:m="Some-URI">
      <GetQuotesResult>18.0</GetQuotesResult>
    </m:GetQuotesResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

C. UNIVERSAL DESCRIPTION, DISCOVERY, AND INTEGRATION—UDDI

Because Web Services can be described, they can also be discovered using UDDI. UDDI is a Web-based directory where businesses can register information about themselves, their services, and detailed technical information in an XML format. One of the main goals of UDDI is to enable e-commerce by defining Web Services offerings and interoperability requirements in a standardized format. As a human metaphor, UDDI is often referred to as Yellow Pages for e-commerce because it is used to find or discover information about potential business relationships based on business name, category of services, products, and Web Services offered. Early on, most analysts expected UDDI to be used primarily in large public registries. While public registries are common, as Web Services matured there has also been strong growth in private UDDI registries located behind firewalls. Reasons behind private UDDI growth include security, well-understood interoperability, and utilizing Web Services as a solution to consolidate corporate functionality. UDDI registries are often also found in trusted, invitation-only trading partner relationships.

In addition to Yellow Pages, UDDI also contains:

- **White Pages.** A more humanly readable listing of information about a business providing listed services
- **Green Pages.** In-depth technical documentation pertaining to specific services

CREATING WEB SERVICES IN PB10

Implementing Web Services with Sybase PowerBuilder 10 is surprisingly simple. Sybase PowerBuilder 10 implements Web Services through the Non-Visual Object (NVO) interface. PowerBuilder Web Services employ stateless PowerBuilder NVO components that can be continually re-invoked by different clients connected to Sybase EAServer—or as is often the case with Web Services, different peers. A single NVO component relates to a single service.

Residing on EAServer, stateless PowerBuilder NVOs are lightweight, efficient, and reusable; the requisite information the NVO needs to run must be supplied at the time it is called, rather than through a multi-stage initialization process. PowerBuilder NVOs are a powerful encapsulation of the necessary functionality to publish rich database information through the interoperability of Web Services. Web Services is an efficient way to reuse an organization’s investment in PowerBuilder development in a service-oriented architecture implementation.
For PowerBuilder-based Web Services, use one of PowerBuilder's EAServer Component wizards:

- **EAServer Component Target Wizard.** Used for new user objects in a new application
- **EAServer Component Object Wizard.** Used for existing application objects

In the component wizards, the option: ‘Expose the component as a Web Service’ needs to be selected. The component wizards manage the information about the component including the component name, the library, the component type, EAServer profiles, and packaging details. The wizard then creates a generator for deploying the component to EAServer. After exposing the PowerBuilder NVO as a Web Services component, the component needs to be deployed to EAServer 5.x or greater as a Web service.

---

**REGISTERING WEB SERVICES IN EAS 5.X WITH THE WEB SERVICES TOOLKIT**

To manage Web Services, Sybase EAServer includes the Web Services Toolkit. The Web Services Toolkit is a full-featured set of tools for creating, registering and managing Web Services in EAServer. Standard installations of EAServer include the Sybase Web Services Toolkit. The toolkit supports standard SOAP, WSDL, and UDDI. It includes tools for WSDL document creation, client generation, UDDI registration/publication, and SOAP management.

Within the Web Services Toolkit is the Sybase Management Console. The Management Console is a J2EE Web application for viewing and managing Web Services contained in EAServer. The Management Console configures EAServer-resident Web Services and manages the public availability of the services, including publishing the services to external public registries. Once published, an EAServer-based Web service can be invoked remotely over HTTP and HTTPS protocols.
In addition to publishing Web Services, PowerBuilder 10 applications can also read or consume Web Services information. Public or corporate Web Services identified through explicit addressing or UUDI search and discovery can be consumed using PowerBuilder 10's SOAP and WSDL capabilities. A PowerBuilder application can request, receive, respond to, and integrate information from a myriad of local and remote sources by acting as a Web Services client.

Building Web Services client capability into a PowerBuilder application requires the following steps:

- Identifying Web Services to be consumed and creating proxies using the Web Services Proxy Wizard
- Aliasing the proxy names to account for XML and SOAP case sensitivity
- Deploying the proxies to an application's PBL
- Adding the pbsoapclient100.pbd to the PowerBuilder target
- Writing PowerScript code to invoke the identified Web Services using SOAP
FINDING WEB SERVICES WITH THE WEB SERVICES PROXY WIZARDS

Discovery is a key feature of Web Services technology; WSDL files either locally located or residing in UDDI registries identify Web Services. PowerBuilder has two Web Services Proxy Wizards to identify and build connections to external Web Services:

- **Web Services Proxy Wizard.** Generates PowerScript proxies to call XML methods.
- **JSP Web Services Proxy Wizard.** Generates proxy overrides for custom beans to call XML methods.

The Proxy Wizards provide two methods for selecting WSDL files. The first method uses the wizard to either access an explicit WSDL file on the local drive, or reference a remote URL containing a WSDL file. The second method is to find a registered WSDL file using the UDDI search feature of the wizard.

The Web Services Proxy Wizard creates a named proxy for each Web service; the proxy identifies the WSDL file, a Web service, the port, and the target library to contain the proxy. Once a WSDL file is found, the Web Services Proxy Wizard prompts the user to select the specific Web service and port for the service connection and creates a proxy generator object in the target library. After the proxy has been created, the individual proxy properties can be modified.

The Web Services Proxy Wizard can be used to access a local file or a specific remote WSDL URL. This non-discovery approach is typically used during testing or in a service-oriented architecture scenario where the location of the WSDL file is known.

When the Web Services Proxy Wizard is used to search a UDDI registry to find a WSDL file, it uses the discovery capabilities of Web Services. A wealth of public UDDI registries is available containing a multitude of services. Registries can be searched by business name and service name. These public registries are an effective way to avoid duplicating development work by finding a specific source of Web Services-based information that might otherwise require in-house development. A downside of public registries is the lack of guaranteed permanence.
FINDING WEB SERVICES WITH THE UDDIPROXY POWERBUILDER EXTENSION

UDDI registries can also be programmatically searched using the UDDIProxy PowerBuilder extension class. This API approach involves more programming than the Web Services Proxy Wizard, but it can build additional flexibility into a Web Services-enabled application. The UDDIProxy has the following methods:

- **SetInquiryUrl.** Sets the UDDI inquiry URL.
- **setOption.** Sets UDDI search options for match precision, case sensitivity, result sort order, and the maximum number of rows returned.
- **findBusiness.** Finds business items using business names in a UDDI search.
- **getBusinessDetail.** Gets business details using a business key that is typically obtained from the findBusiness function.
- **FindService.** Finds service details using a service name.

More information and programming examples for the UDDIProxy extension class can be found in the PowerBuilder Extension Reference.

ALIASING PROXY NAMES FOR XML AND SOAP METHODS

Unlike PowerBuilder, XML and SOAP are case sensitive. For PowerScript code to call XML methods correctly, each method in the proxy needs an alias. This is accomplished with the ‘alias for’ directive. The following example shows how to alias a PowerBuilder proxy to a correctly cased SOAP method:

```powerbuilder
function real getquote(string ticker) alias for getQuote(xsd:string symbol)#
return xsd:float StockPrice@urn:xmethods-delayed-quotes@SoapAction
```

SOAP CONNECTIVITY USING PBSOAPCLIENT100.PBX

PowerBuilder 10 provides SOAP functionality with the pbsoapclient100.pbx file. Internally, SOAP requires some moderately complex implementation functionality such as serializing and deserializing passed data. The pbsoapclient100.pbx file provides a high-level interface that isolates the PowerBuilder developer from the low-level implementation details.

The pbsoapclient100.pbx file is part of a standard PowerBuilder installation and resides in the Shared/PowerBuilder directory. The pbsoapclient100.pbx file does not need to be copied to a different location, but it does need to be in the application’s search path and deployed with the client executable.

The pbsoapclient100.pbd contains two objects:

**SoapConnection**—Manages SOAP connections

- Instantiates the proxy object and populates SOAP options
- Adjusts SOAP options such as log file, authorization, proxy etc.

**SoapException**—Manages error handling within SOAP connections

- SoapException is inherited from the PowerBuilder RuntimeError object
- Errors occurring in the execution of a method within a Web service are converted to SoapException objects and thrown to the calling script
The following PowerScript code invokes the actual Web service:

```powerbuilder
SoapConnection cnn // Declare
SoapConnection dm_tw2004_n_ws_demods proxy_obj // Declare proxy

// Create proxy object
cnn.CreateInstance(proxy_obj, "dm_tw2004_n_ws_demods")

String s_returnValue

try
    s_returnValue = proxy_obj.getds() // Invoke Web Service
    // use the return value
    ...

catch ( SoapException e )
    messagebox ("Error", "Cannot invoke WS") // error handling
end try

destroy cnn
```

## XML to PowerBuilder Type Mapping Table

<table>
<thead>
<tr>
<th>XML Type</th>
<th>PB Type</th>
<th>XML Type</th>
<th>PB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>decimal</td>
<td>decimal</td>
</tr>
<tr>
<td>byte</td>
<td>int</td>
<td>float</td>
<td>real</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>uint</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>uint</td>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>int</td>
<td>long</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>ulong</td>
<td>dateTime</td>
<td>dateTime</td>
</tr>
<tr>
<td>long</td>
<td>longlong</td>
<td>base64</td>
<td>blob</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>longlong</td>
<td>base64Binary</td>
<td>blob</td>
</tr>
<tr>
<td>integer</td>
<td>longlong</td>
<td>hexBinary</td>
<td>blob</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>longlong</td>
<td>negativeInteger</td>
<td>longlong</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>longlong</td>
<td>positiveInteger</td>
<td>longlong</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>string</td>
<td>gYear</td>
<td>string</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>string</td>
<td>gMonth</td>
<td>string</td>
</tr>
<tr>
<td>gDay</td>
<td>string</td>
<td>duration</td>
<td>double</td>
</tr>
<tr>
<td>anyURI</td>
<td>string</td>
<td>QName</td>
<td>string</td>
</tr>
<tr>
<td>NOTATION</td>
<td>string</td>
<td>normalizedString</td>
<td>string</td>
</tr>
<tr>
<td>token</td>
<td>string</td>
<td>language</td>
<td>string</td>
</tr>
<tr>
<td>NMTOKEN</td>
<td>string</td>
<td>NMTOKENS</td>
<td>string</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
<td>NCName</td>
<td>string</td>
</tr>
<tr>
<td>IDREF</td>
<td>string</td>
<td>IDREFS</td>
<td>string</td>
</tr>
<tr>
<td>ENTITY</td>
<td>string</td>
<td>ENTITIES</td>
<td>string</td>
</tr>
<tr>
<td>ID</td>
<td>string</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
POWERBUILDER TO XML DATA TYPE MAPPING
When moving information between PowerBuilder and XML, it is important to understand the
differences in data types. The following table demonstrates the mapping of Web Services XML
data types to PowerBuilder data types:

CONSUMING WEB SERVICES IN PB10—JSP CLIENT
PowerBuilder 10 also supports accessing Web Services from a JSP. This involves using the JSP
Proxy Wizard and creating a JSP page.

GENERATE CUSTOM TAG USING JSP WEB SERVICES PROXY WIZARD
Similar to the Web Services Proxy Wizard, the JSP Web Services Proxy Wizard collects information
such as location of the WSDL file, service, and port settings. It also allows user-specified
overrides for custom bean name, Java class name, Java package name, Tag Library Descriptor
(TLD) name, Jar name, output variables, and operation selection within a service.

The JSP Web Services Proxy Wizard outputs a TLD file, Java source to process the custom tag,
and compiled Java class files that can be deployed to a JSP container or used as standalone Java
classes.

WRITE JSP PAGE
When the JSP Web Services Proxy Wizard has generated the output, go to the system tree and
choose the custom tag representing the Web service to drag and drop on the JSP. The goal is to
identify to the JSP container the custom tags to be used in the page. The next steps are adding the
Tablib directives, specifying the input and output arguments (all arguments must be represented as
objects because they are stored in the pageContext of the JSP container), invoking the custom tag,
and handling errors. The final step is deploying the JSP to Sybase EA Server or any JSP server.

Below is a sample JSP page for consuming a Web service:
<%-- Add custom tag to this page --%>
<%@ taglib uri="WEB-INF/tlds/pkg_customer_n...tld" prefix="cust" %>

<%-- Define variable to pass into custom tag --%>
<%! Short custid = new Short((short)107);
org.omg.CORBA.StringHolder address = new
org.omg.CORBA.StringHolder();
... %>

<%-- Run custom tag --%>
<cust:of_getcustomer address="<%= address %>">
cust_fname="<%= cust_fname %>">
.. />

<%-- Print return results --%>
<%= address.value %></TD>
<%= pkg_customer_n_customerService_of_getcustomer_returnValue %>
SUMMARY

PowerBuilder NVOs can be deployed and exposed as Web Services directly from the PowerBuilder IDE. By coupling PowerBuilder with Sybase EAServer, this is easily accomplished and can move a body of tight, legacy code into a service-oriented architecture for corporate or public consumption.

Sybase PowerBuilder 10 can also consume Web Services using PowerBuilder or JSP clients. The clients pull information from external Web Services hosted by different types of servers including EAServer, .NET, WebLogic, WebSphere, and other application servers into an existing PowerBuilder application to further increase the application’s utility.

WEB SERVICES—A NATURAL EXTENSION OF POWERBUILDER

PowerBuilder is a streamlined rapid application development environment for creating robust database applications. As PowerBuilder matured and added Web deployment capabilities, development shops were able to leverage their PowerBuilder code, initially used in a client/server architecture, into Web-based applications. The PowerBuilder Non-Visual Object (NVO) is a flexible approach that cleanly adapts PowerBuilder code to Web-based client/server applications. With Web Services, PowerBuilder NVO code moves into the realm of Web-based peer-to-peer architectures.

Companies with a large repository of PowerBuilder-based application code have a significant investment in its development; it has been successively refined and perfected through multiple development passes. PowerBuilder’s Web Services capability further leverages this investment into the arena of service-oriented architecture. Configuring PowerBuilder to provide and consume Web Services is straightforward; it is a natural extension of an existing body of PowerBuilder code, a variation on a theme of targeting business applications where they will have the most impact.

Web Services are flourishing; this platform independent architecture is playing an important role in bringing the Internet to a new stage of functionality. Web Services survived a level of early hype and overzealous prediction that could have been the kiss of death to a lesser technology. Web Services technology is sound and proven; the backlash of doubt resulting from the early hype tempered Web Services adoption with a fundamental practicality. In addition to the originally envisioned free-for-all exchange of information, there has been an explosive growth in private UDDI servers at corporate, trading partner, and trading alliance levels. These Web Services perform real tasks, supplying and consuming valuable information in trusted relationships.

Rather than being used for research on college campuses, PowerBuilder is a tool for rapidly developing solid, business-oriented applications with serious database back ends, and in this role it is a natural complement to the practical side of Web Services implementations.