Java DataSet

Markus Lorez
University of Applied Sciences, Rapperswil
Oberseestrasse 10
CH – 8640 Rapperswil
Markus.Lorez@hsr.ch

Alain Schneble
University of Applied Sciences, Rapperswil
Oberseestrasse 10
CH – 8640 Rapperswil
a.s@realize.ch

ABSTRACT

Today's applications are required to distribute data beyond a company's intranet and access services located all over the world. XML and Web Service technologies provide portable solutions in a heterogeneous Internet environment. Still, the data has to be interpreted according to schema definitions and transformed into an appropriate intra-application representation. The .Net DataSet can represent relational data and exchange it using the XML syntax. The XML schema used by the DataSet (DiffGram) to map data to XML is proprietary. Using the XML data as is on another platform requires additional parsing and interpretation. The goal of this project was to re-implement the .Net DataSet in Java to provide seamless interoperability between .Net Web Services using DataSets and Java Web Service clients/consumers. This paper discusses the need for a Java DataSet and the problems that arose during the reimplementation. Further, it summarises the Java implementation and the seamlessness/transparency of the Java DataSet integration.

Keywords

Toolkit-Interoperability, .Net Web Services, DataSet, XML, Interoperability with JavaJAX-RPC/Axis, Data-Centric Applications, Porting Components from .Net to Java

1. INTRODUCTION

Almost every application needs to store data. Relational databases are still the most common solution for storing information at least for data-centric business applications. These applications often directly manipulate this data and a relational representation is appropriate or even desired (e.g. to present the data in a table/grid). There is no need for a complex object oriented domain model for such applications because it will not offer any benefits – a relational model is sufficient.

Another characteristic of data-centric applications is the requirement to work with disconnected data. But working with disconnected data poses the problem of concurrent data modification. This in turn requires the application being aware of modifications done on another's behalf.

In terms of interoperability, Web Services are currently the state-of-the-art for building distributed systems. Web Services typically use SOAP [Soa] as message protocol, which itself relies on XML. These technologies allow to access services built on one platform (e.g. .Net) to be accessed by clients built on a different platform (e.g. Java). But interoperable services have to exchange the data passed in messages in a portable format as well (i.e. XML).

.NET Technologies'2005 *conference proceedings,*
ISBN 80-86943-01-1
Copyright UNION Agency – Science Press, Plzen, Czech Republic

The Microsoft .Net platform easily allows developers to build interoperable distributed systems, because technologies such as Web Services and XML are an integral part of the framework. The framework further offers an applicable concept called DataSet. The DataSet is capable of holding an in-memory representation of relational data. It can even be used in combination with Web Services as data exchange container because it allows serialisation and deserialisation to and from XML.

But there is a problem when accessing a .Net Web Service using DataSets from another platform (e.g. Java), because the platform-dependent DataSet construct is not available. Even though DataSets use XML as serialisation format, interpreting and reconstructing the relational model is a complex, error-prone and time-consuming task.

This would require building interoperable .Net Web Services without DataSets. However, DataSets provide a practical and applicable solution and can help to solve some reoccurring problems like the concurrency issue when working with disconnected data. Because many .Net Web Services are (and will be) built using DataSets, there is a need for an easy access to these services from another platform.

For the reimplementation of the .Net DataSet, Java 5 has been chosen because the Java Platform has proven to be a robust environment for distributed business applications as well as .Net.

Microsoft .Net DataSet

The DataSet is able to hold an in-memory representation of relational data. It can be compared to a relational database: it allows the definition of a relational schema (tables, columns) and the storage of data according to this schema. The DataSet even supports constraints (i.e. unique/foreign key constraints and allow/deny DBNull values). This makes it an ideal replacement for a domain model in data-centric applications.

The DataSet is meant to be passed through different software layers, from the data access up to the user interface layer. The .Net framework supports this approach by offering classes that enable two-way communication between the DataSet and database. Further, a DataSet can directly be bound to user interface components supporting data binding (e.g. to a table/grid).

What makes the DataSet such a usable data container for disconnected data is its capability to store different versions of the data (i.e. original, current and proposed). It thus implements some kind of unit of work pattern [Fow02], because it allows a client to modify data and retransmit the modified DataSet back to the server once all update operations are completed by the client. By including the original data as well, it can easily be determined which data were already modified by another client in the meantime.

When DataSets are used in WebServices, they are passed as XML payload including both the schema and the data. The schema is described by an (extended) XML Schema. The data is represented by an XML grammar called DiffGram, which supports the representation of current and original data as well as error information concerning the data.

2. JAVA DATASET

As described earlier, a platform-independent implementation of the .Net DataSet is highly desirable. Actually, there are two different possible ways to reach this goal. The first alternative would be to port the DataSet (or the .Net framework in its entirety) to different native platforms resulting in a number of several platform-dependent versions. In fact, this approach is already realised to some extent by the Mono project [Mon]. The second alternative would be to port the DataSet once to a platform-independent framework, such as Sun's Java. The latter approach is the ultimate goal of the Java DataSet project.

Goal

On a lower level, there are several goals and requirements for a DataSet reimplementation in Java. Since the project was limited to 16 weeks, the desired functionality had to be adapted to that time restriction.

We included everything that is essential to use the DataSet in a client. This includes the components of the relational data model (such as tables, rows, columns, constraints and relations), row state handling, (XML-) serialisation and deserialisation and the GetChanges method. Other DataSet components, however, are not vital in client use (such as DataViews or DataAdapters).

Since most developers who implement a Java client using DataSets are familiar with the .Net DataSet, the syntax should be as near as possible to Microsoft's DataSet. Luckily, C# and Java (especially version 5.0) are quite similar except for a few language concepts.

Another requirement was that the installation of the Java DataSet library should be kept as simple as possible. Therefore, third-party libraries should be avoided. The Java DataSet itself uses no additional libraries. For Web Service access, however, Apache Axis is used.

Implementation

Before porting a highly complex construct – such as the .Net DataSet – to another platform, thorough analysis of the original is indispensable. Unfortunately the (otherwise very good) documentation by Microsoft is only helpful to some extent because it is designed to help application developers using the DataSet. It is not suited, though, to support a developer intending to dissect the DataSet's internals and re-implement it on another platform. As a consequence, the DataSet's internal mechanics must be discovered by other means, such as own tests or even analysis of the IL code (Intermediate Language, comparable to the ByteCode of Java).

Once these difficulties have been overcome, the implementation of the Java DataSet is quite straightforward. The major differences to the original are due to the divergence of the Java and C# languages and their frameworks, respectively. One important difference in usage is the direct invocation of getter and setter methods in Java. Java properties are

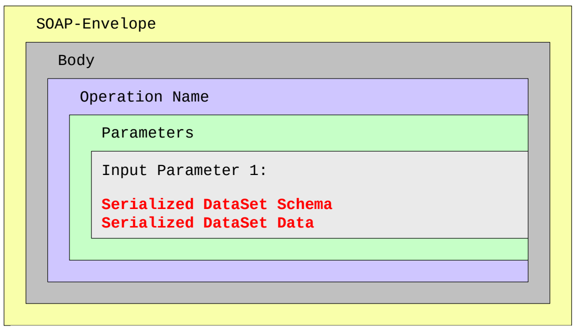

Figure 1. SOAP-Envelope of an RPC-oriented operation call with a DataSet as operation parameter.

merely a naming convention as opposed to being built into the language itself as in C#.

When it comes to porting software to a different platform there is a regular issue: data types. It is a peculiarity of the Java framework that there are no unsigned data types. Therefore, one has to implement a custom mapping mechanism to map the upper half of the unsigned C# data type's range to the negative part of the corresponding Java type, leading to a fair amount of additional complexity. The simpler approach used in the Java DataSet is to use the next bigger type class, allowing the whole unsigned value range to fit smoothly into the positive half of the Java type.

Third issues are access modifiers (public, protected, internal, private). In C#, access scope is based on assembly structure whereas logical grouping is provided by namespaces. As a consequence, access scope and logical grouping are orthogonal. In Java, both access scope and logical grouping are realised by packages. In the DataSet, the different classes collaborate tightly by calling members of other DataSet library classes, which implies that all classes must be located in the same package (resulting in a rather large package).

Outlook

The Java implementation is far from complete. As mentioned above, there are several components in the original Microsoft DataSet that are not yet implemented in the Java port. However, for now it is possible to use the DataSet as a general data (transport) container as well as in Web Service to client communications.

Additionally, there is a usage scenario of the Java DataSet in Java GUI applications. Since the DataSet provides a tabular data structure, it is an ideal table model providing data values to a JTable. Only an additional small intermediate layer between DataSet and JTable would be necessary, resulting in a setup similar to the combination DataSet/DataGrid in .Net.

```
<soap:Envelope xmlns:soap="..." ...>
  <soap:Body>
    <UpdateAccounts xmlns="...">
    <AccountsParameter>
      <xs:schema id="AccountsDataSet" ...>
        <xs:element name="AccountsDataSet"
                    msdata:IsDataSet="true">
          <!-- Accounts DataSet
               schema definition -->
        </xs:element>
      </xs:schema>
      <diffgr:diffgram ...>
        <!-- account data -->
      </diffgr:diffgram>
    </AccountsParameter>
    </UpdateAccounts>
  </soap:Body
</soap:Envelope>
```

Figure 2. SOAP message transmitting a DataSet as operation parameter.

3. CONSUMING .NET WEB SERVICES FROM JAVA

.Net Web Services rely on SOAP[1] as XML message protocol. SOAP is currently wide-accepted and there are many implementations available. In Java, most "RPC-oriented" implementations follow the APIs and conventions defined by JAX-RPC [Jax]. A feasible SOAP implementation conforming to JAX-RPC is Axis [Axi]. Using Axis, a Java client can consume a (.Net) Web Service without problems as long as standard data types (like xs:string) are used as input/output parameters for Web Service operations. Using custom data types (like DataSets) poses some problems: a custom serialisation and deserialisation has to be implemented and the SOAP implementation needs to be extended to offer transparent usage.

Passing DataSets in SOAP Messages

A common approach to building a .Net Web Service that uses DataSets as data exchange containers is to follow the RPC-oriented invocation style[2]. For example, an operation could involve updating the database to reflect the modified data contained in a DataSet. This means that DataSets have to be passed to the Web Service as operation parameter (or returned as the operation's return value). Figures 1 and 2 show an example of an RPC-oriented SOAP Message – emitted by a .Net Web Service client – containing a

[1] Since SOAP Version 1.2, the term SOAP has two expansions – Service Oriented Architecture Protocol and Simple Object Access Protocol – to reflect the different ways in which the technology can be interpreted.

[2] .Net by default uses "Document" as message and "Literal" as serialisation format [Rpc]. This paper uses the term RPC-oriented independently of the underlying message format because "Document" is a superset of "RPC" and it can also be used to mimic an RPC-oriented invocation style – which in fact is what .Net does by default.

DataSet as operation (input) parameter. The response sent by a .Net Web Service looks very similar. The default behavior of a .Net Web Service passing DataSets is to include both the DataSet schema and data, whereas the data is represented as DiffGram.

The interoperability issue does not primarily lie in the "generic" message parts being transmitted because they (should) conform to the W3C SOAP recommendation but rather in the custom data included as parameter or return value. Basically, a SOAP implementation is aware of simple types and some array encoding styles. Custom types can sometimes be mapped to classes (e.g. Java Bean classes) automatically using tools (e.g. JAXB in Java or the WSDL2Java utility from Axis). But this approach is often not applicable to complex data types. The .Net DataSet falls into this category as both the schema and the data have to be interpreted and a "simple object representation" would not suffice. The following section presents the solution realised by the Java DataSet implementation on top of Axis.

JAX-RPC/Axis and Transparent Usage

Axis implements the JAX-RPC API and offers the ability to extend the default Java-to-XML type mapping using JAX-RPC interfaces. For custom data types like the DataSet, specialised serialisers and deserialisers have to be implemented to enable Axis to transform the XML representation to a Java object and vice versa. Thus, the DataSet requires a DataSetSerializer and DataSetDeserializer that are aware of this transformation process. Serialisers and deserialisers will not be instantiated directly by Axis because it delegates this work to factory classes: DataSetSerializerFactory and DataSetDeserializerFactory.

The DataSetDeserializer is event driven – it receives SAX-Events caught and forwarded from the Axis infrastructure. Axis calls the appropriate factory to obtain a deserialiser instance whenever it can find a registered XML type. Similarly, it calls the appropriate factory to serialise a registered Java type using the returned serialiser.

A custom type mapping can be registered using the TypeMappingRegistry. Normally, registering a custom type mapping involves the following steps:

1. Get a reference to the default type mapping registry
2. Instantiate serialiser and deserialiser factories
3. Register a new type mapping between the Java class and XML-Type and specify both factory instances

The Java code needed to set up such type mapping is shown in Figure 3. When creating an ASP.Net WebService, the WSDL document defines custom types for each operation's parameters and/or return value. This involves registering a custom type mapping for each DataSet-type parameter and return value.

```
ServiceFactory sf = ServiceFactory
                    .newInstance();
Service webSvc = sf.createService(url,
                 qWebServiceName);
TypeMapping tm = webSvc
                 .getTypeMappingRegistry()
                 .getDefaultTypeMapping();
tm.register( DataSet.class,
         qualifiedXMLTypeName,
         new DataSetSerializerFactory(),
         new DataSetDeserializerFactory() );
```

Figure 3. Registering a custom type mapping between Java and XML.

Usage Example

By implementing custom serialisers/deserialisers and registering the appropriate type mappings, the invocation of Web Service operations receiving and returning custom data types is transparent to the caller. The listing in Figure 4 provides an example of a Web Service call invocation returning a DataSet instance to the caller, assuming that the correct type mapping was registered.

```
ServiceFactory sf = ServiceFactory
                    .newInstance();
Service webSvc = sf.createService(url,
                 qWebServiceName);

Call call = webSvc.createCall(qPortName,
              qOperationName);
DataSet dataSet = (DataSet) call
                  .invoke(null);
```

Figure 4. WebService call returning a DataSet.

4. CONCLUSION

Despite its incompleteness, the current Java DataSet implementation allows the simple and transparent exchange of data between .Net Web Services and Web Service consumers in Java. The development of a Web Service consumer is highly simplified by a ready-to-use Java DataSet.

5. REFERENCES

[Fow02] Fowler, Patterns of Enterprise Application Architecture, 2002.

[Soa] W3C SOAP Recommendation, http://www.w3.org/TR/soap/

[Mon] Mono Project, http://www.mono-project.com

[Axi] Axis Project, http://ws.apache.org/axis/

[Jax] Java API for XML RPC, http://java.sun.com/xml/jaxrpc/index.jsp

[Rpc] RPC/Literal and Freedom of Choice, MSDN, /library/en-us/dnwebsrv/html/rpc_literal.asp