



xTier™ Marshal Service

The xTier™ Marshal Service provides cross-paradigm data interchange for Java and .NET.

Key Benefits:

- Up to 25x faster than native Java and .NET serialization
- High-performance data encoding and decoding
- High-performance marshalling and de-marshalling
- Cross-paradigm, data-interchange between Java and .NET is handled automatically and safely.

Marshalling Overview

Marshalling can be defined as the process of packing one or more items of data into a message buffer prior to transmitting that message buffer over a communication channel. The packing process not only aggregates data values which may be stored in non-consecutive memory locations but also converts data of different types into a standard representation agreed upon with the recipient of the message.

Marshalling is usually required for cross-platform data exchange. For instance, it can be used for data exchange between Java and .NET when passing input/output parameters between these platforms, and allows developing multi-platform applications. The marshalling mechanism compensates for the differences between operating systems or platforms, without taking care of formats differences, byte order or other details, which is handled during the encoding/decoding process.

A concept closely connected with Marshalling is the concept of serialization. Serialization is the process that allows objects to be converted to a sequence of bytes (or XML, or a SOAP scheme, or other format), which can then be stored to disk or transmitted across the network where the de-serialization process uses this encoded data to recreate the object. This serialized object can be used in a variety of different ways, as the restoration of the object can be performed on the application's next start, on another machine, or by another application, depending on the task. There is an important distinction to be drawn when comparing marshalling and serialization; notice that marshalling can be used for cross-platform data exchange, while serialization can not.

Marshaling is typically used in two ways: persistence and data interchange. Persistence stores the encoded information with some non-volatile mechanism for future use (for instance, storing data between application executions, data archiving, etc.). When used for data interchange marshaling is a very versatile facility. If the application takes the form of an N-tier solution, it will need to transfer information from client to server, likely using a network protocol such as TCP. To achieve this data interchange the data structure would be marshaled into a series of bytes that can then be transferred over the network.

Since marshalling and serialization are related concepts (although remember that they are distinguished because serialization is not intended for cross-platform interchange), the following section will provide a brief review of the generic serialization solutions provided by the Java and .NET platforms, and will then describe the xTier™ Marshalling Service features, functionality, usage and advantages.

Java Serialization

Serialization provides the ability to converting any object that implements the **java.io.Serializable** interface into a byte sequence, from which it is possible to entirely restore the initial object. The stream functions as a container for the object. Its contents include a partial representation of the object's internal structure, including variable types, names, and values. Information stored in the container can later be used to construct an equivalent object containing the same data as the original.

The serializability of a class is enabled by the class implementing the **java.io.Serializable** interface. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable. To use the basic serialization mechanism an **OutputStream** is needed, which should be passed to the **ObjectOutputStream**. Then the **writeObject()** method should be invoked to serialize the object and send it to the output stream. To restore the object pass an **InputStream** to the **ObjectInputStream** and invoke the **readObject()** method. In this case the default Java serialization/de-serialization protocol is used.

In the case that the default serialization/de-serialization protocol is not sufficient, the developer, by implementing the **Externalizable** interface instead of the **Serializable** interface has the ability to explicitly control the serialization process. The **Externalizable** interface extends the **Serializable** interface by adding two methods, **writeExternal()** and **readExternal()**, which are automatically called during serialization and de-serialization. The developer provides an implementation of these methods which then performs the class serialization and de-serialization according to their needs, however, it is important to note that although the underlying protocol serializing the class members will now be using a proprietary encoding/decoding scheme, it will still be able to operate in a Java-only environment, thus serialization can not be used for cross-platform data exchange.

Serialization in .NET

The .NET platform provides several serialization solutions that provide methods to serialize objects into a binary format, a SOAP format, or into XML. The XML format is produced by using the **System.Xml.Serialization.XmlSerializer** class. The SOAP and binary formats are produced by using classes under the **System.Runtime.Serialization.Formatters** namespace. In order to use a formatter, the class should be marked using the **Serializable** attribute. Developers are able to create their own formatters if needed.

xTier™ Marshal Service

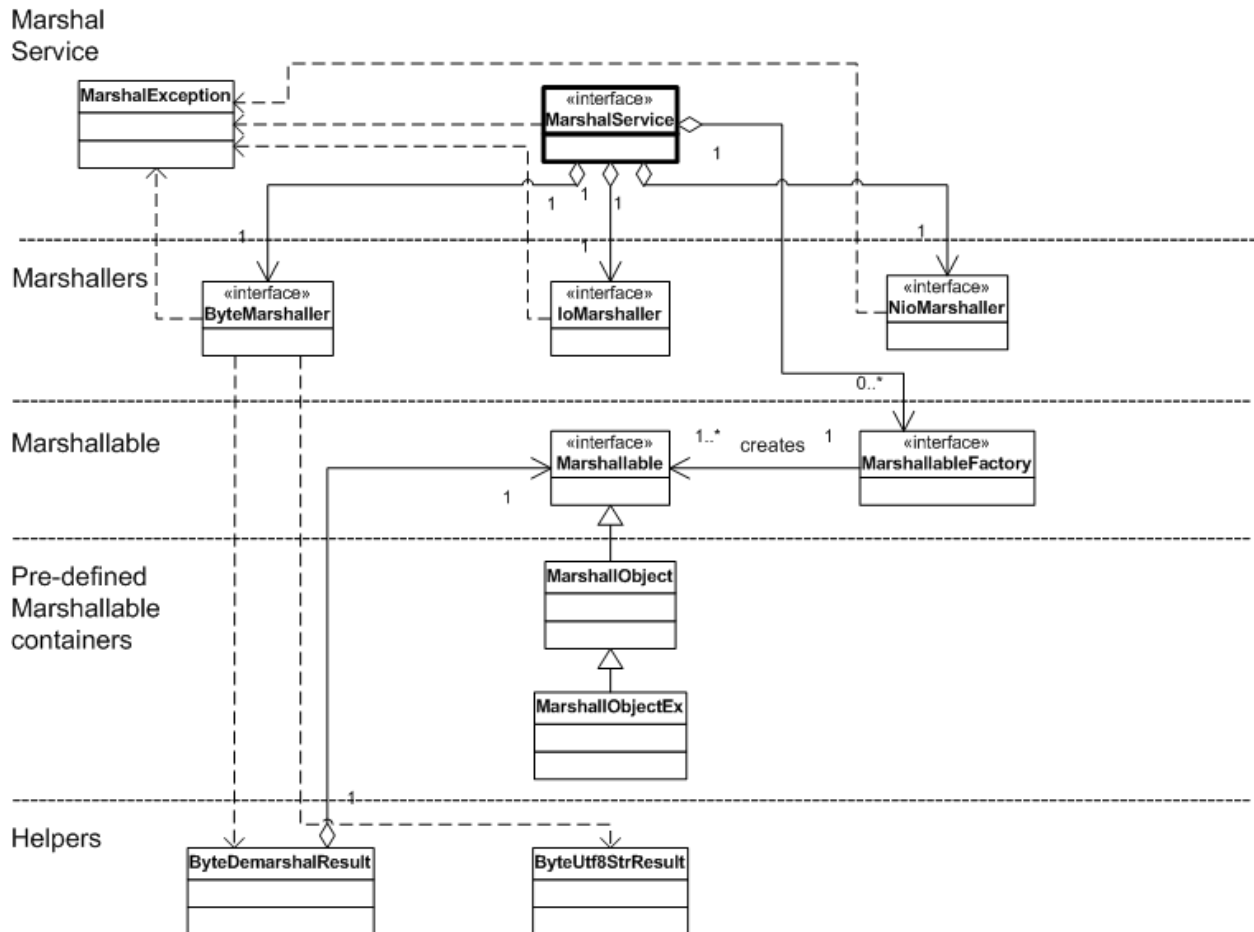
The xTier™ Marshal Service enables high-performance data exchange in the following configurations:

- between Java and .NET
- between Java and Java
- between .NET and .NET

It is important to note that the xTier™ Marshal Service is network protocol, component model and generally system architecture agnostic as it only concerns itself with transforming the native data presentation to and from a certain format. For instance, it can be used to generate payloads for SOAP envelopes, binary email attachments or to encode "Value Objects" in an EJB infrastructure.

Components

In order to become familiar with the xTier™ Marshal Service, the best place to start is to consider its common service's and list of components (interfaces and classes) that the xTier™ Marshal Service consists of. The following illustrates the class diagram representing the service structure:



The xTier™ Marshal Service consists of the following components:

Marshallers:

- **IoMarshaller** – this interface defines the API for the IO marshaller.
- **NioMarshaller** – this interface defines the API for the NIO marshaller.
- **ByteMarshaller** – this interface defines the API for the byte marshaller.

Pre-defined containers which support marshalling:

- **MarshalObject** – this class defines the generic container that supports marshalling.
- **MarshalObjectEx** – this class defines the generic container that support marshalling with support for Externalizable.

The **MarshalService** interface defines the main API for the xTier™ Marshal Service. It provides methods for obtaining marshaller instances, and registering and un-registering marshallable factories. The **ByteDemarshalResult** class – is a helper class used by **ByteMarshaller** to combine de-marshalled objects with the new offset. The **ByteUtf8StrResult** class – is a helper used by **ByteMarshaller** for string decoding. The **MarshalException** class defines the exception used throughout the marshal service. The **Marshallable** interface defines the API for marshallable objects and the **MarshallableFactory** interface defines the API for marshallable factories. The following sections examine these components in detail.

Marshallers

The xTier™ Marshal Service provides three types of input/output "media" for marshalling and de-marshalling:

- **byte[]** - provided by the **ByteMarshaller** interface.
- **InputStream** and **OutputStream** - provided by the **IoMarshaller** interface.
- **ByteBuffer** - provided by the **NioMarshaller** interface.

Marshaller instances can be obtained from the Marshal Service using these methods: **getByteMarshaller()**, **getIoMarshaller()** and **getNioMarshaller()**. Each formatter provides two types of functionality: encoding/decoding and marshalling/de-marshalling.

Encoding and Decoding

The encoding/decoding functionality provided by the formatters provides basic encoding and decoding for primitive and reference types. This is provided so that the developer can create a custom marshalling and de-marshalling protocol if the one provided by xTier™ is not suitable. These encoding and decoding methods provide all the necessary basic type transformations in a cross-platform fashion so that the encoding and decoding is Java or/and .NET ready right out-of-the-box.

Each of the 3marshallers provides cross-platform encoding and decoding for the following primitive types:

- signed 8-bit, 16-bit, 32-bit and 64-bit integer
- 16-bit character
- 32-bit and 64-bit real
- boolean

and the following reference types:

- String
- Date

In order to provide an example of the use of the encoding and decoding methods, consider the following example that encodes and decodes **boolean** values (please note that this example is Java based) The **encodeBool()** method encodes Boolean values. In the **ByteMarshaller** class the method has the following signature:

```
public int encodeBool(boolean value, byte[] arr, int off)
```

The first argument is the **boolean** value to encode. The second argument is the array of bytes where to put encoded value. The third **int** argument is an offset. The method writes the encoded value to the array by the offset index. Note that the method returns not the array with the encoded value (as the developer already holds a reference to this array as he or she passed it as an argument), but the **int** value representing the new offset in this array. This is a known technique that provides convenience when encoding several primitives into one array. When encoding the next primitive, simply pass the offset value obtained in the previous step.

The **IoMarshaller** and **NioMarshaller** classes encode and decode methods have similar signatures but receive a **java.io.OutputStream** or **java.nio.ByteBuffer** instead of a byte array. The main difference is that there is no to pass any offsets, because the stream and **ByteBuffer** maintain their own offsets. Since there is no need for offsets, these methods do not need to return anything and thus have void types.

The corresponding decoding method for the **ByteMarshaller** is the following:

```
boolean decodeBool(byte[] arr, int off)
```

The arguments passed to this method consist of an array of bytes to decode and the offset in this array. The **IoMarshaller** and **NioMarshaller** methods use a **java.io.InputStream** and **java.nio.ByteBuffer** correspondingly and do not receive an offset, because the stream and byte buffer maintain the offset internally.

Marshalling and De-marshalling

The xTier™ Marshal Service implements a marshalling and de-marshalling protocol that can be used for high-performance object exchange between Java and .NET, Java and Java or .NET and .NET. The fundamental goal of this protocol is to provide an extremely high-performance marshalling and de-marshalling infrastructure. Testing shows that the xTier™ Marshal Service can achieve performance of up to 25x times faster than native Java or .NET serialization. That translates, for example, into up to 1000 times faster performance as compared to SOAP-based marshalling. This high performance ultimately enables the marshal service to be used in near Real-Time (nRT) applications.

In order to be marshallable and de-marshallable an object must implement the **Marshallable** interface. This interface defines a contract that has to be implemented by every marshallable object. The **Marshallable** interface specifies a marshallable object as a map of properties and a unique global ID (GUID). Keys and values in the property map can be of any of the following types (note that all types are reference types; arrays can be primitive):

- | | |
|--|---|
| <ul style="list-style-type: none">• Byte• Short• Integer• Long• Float• Double• Character• Boolean• String | <ul style="list-style-type: none">• Date• Hashtable (keys and values should also be one of these types)• ArrayList (values should also be one of these types)• Marshallable• Java arrays (elements should also be one of these types or Java primitive types) |
|--|---|

As evidenced by this list, the **Marshallable** interface can represent a wide range of objects. A **Marshallable** map can contain values which are maps itself, of arrays, or lists, or another marshallables, and these containers can also include objects of these types enumerated above, and so on. The **Marshallable** interface specifies several methods:

```
public short typeGuid()
```

This method should return a global type UID (GUID) for the marshallable class. As the GUID characterizes a class, it may be provided in the implementation as a static constant, which the **typeGuid()** method should return. Note that all GUID in [**Short.MAX_VALUE**, **Short.MAX_VALUE - 255**] are reserved for internal use by xTier™. User defined GUIDs must be defined outside of this range.

```
public java.util.Map getObjs() throws MarshalException
```

This method obtains a map of objects representing the marshallable state or body. Keys and values should be one of the supported types specified above.

```
public void setObjs(java.util.Map objs) throws MarshalException
```

This method sets the map of objects representing the marshallable state or body. Keys and values should be one of the supported types specified above.

```
public void onMarshal() throws MarshalException
```

```
public void onDemarshal() throws MarshalException
```

These methods are called right before an object is marshaled and right after an object is de-marshalled respectively. Implementations of these methods can be no-op, if no specific actions are required. The implementation can throw a **MarshalException** exception to halt the marshalling process.

Next consider the signature of the methods that perform marshalling and de-marshaling. When using a **ByteMarshaller** the following methods are utilized:

```
byte[] marshalObj(Marshallable obj)
```

```
int marshalObj(Marshallable obj, byte[] arr, int off)
```

```
public ByteDemarshalResult demarshalObj(byte[] arr, int off) throws MarshalException
```

```
public int demarshalObj(byte[] arr, int off, Marshallable obj) throws MarshalException
```

It is important to note that each method has two signatures. In the case of marshalling one signature receives only a marshallable object as an argument and returns an array of bytes representing the encoded object. The second method signature allows passing a byte array and an offset as arguments and returns a new offset value. In this way it is possible to marshal several objects into one array. For de-marshalling there is a similar set of methods. The first **demarshalObj()** signature returns a **ByteDemarshalResult** object. This is a helper container that combines the de-marshalled object and the new offset in the byte array. The **getObject()** and **getOffset()** methods can be used in order to obtain the de-marshaller object and the offset.

IoMarshaller and **NioMarshaller** each provide only one signature for the **marshalObj()** method, which receives **Marshallable** and **java.io.OutputStream** or **java.nio.ByteBuffer** as arguments. Remember, that with streams and byte buffers the offsets are internally maintained. The **demarshalObj()** method has two signature for thesemarshallers, one of which de-marshals a given input stream or byte buffer into the marshallable object provided as an argument.

Pre-defined Marshallables

The xTier™ Marshal Service provides two pre-defined marshallable objects, which implement the **Marshallable** interface, and can be used as the base for custom marshallables to minimize the developer's work: **MarshalObject** and **MarshalObjectEx**. **MarshalObject** defines a generic marshallable container that has an internal Map and provides a set of setters and getters for strongly typed access to the object's properties. To be more exact, each type from the list of supported types has an appropriate getter and setter method. The **onMarshal()** and **onDemarshal()** methods have no-op implementations. Note that the **MarshalObject** implements the **Serializable** interface.

The **MarshalObject** defines 3 public constructors:

```
MarshalObject()
```

```
MarshalObject(int maxCapacity)
```

```
MarshalObject(int capacity, float loadFactor)
```

The first is the default (no-argument) constructor. The second constructor creates a marshal object backed by a map with the given maximum capacity (**int maxCapacity**) and which will never rehash. The final constructor creates an instance of this class with the specified initial capacity (**int capacity**) and load factor (**float loadFactor**). The class also provides

2 protected constructors, which should be used by subclasses to override the class GUID. Note subclasses, based on the **MarshallableObject**, must override the GUID for the class.

Another pre-defined **marshallable** container, **MarshallableObjectEx**, defines the generic marshallable container adding support for **Externalizable** to its super class **MarshalObject**. This class implements the **readExternal()** and **writeExternal()** methods from the **Externalizable** interface to save and restore its content to the specified object stream.

For user-defined marshallable objects the developer has to provide an implementation of the **MarshallableFactory** interface that allows the marshal service to internally create the user-defined objects. This interface specifies only one method:

```
public Marshallable tryNewInstance(short typeGuid)
```

The implementation of this method should create a new marshallable instance for a given type GUID. This method must return null if this factory does not handle the given type GUID. Note that the developer does not have to create a marshallable factory for the pre-defined marshallable containers (**MarshalObject** and **MarshalObjectEx**).

User Defined Formatters

If the xTier™ pre-built marshalling and de-marshalling protocol is not suitable or does not cover all of the developer's needs for some reason of other, the developer can implement their own custom marshalling and de-marshalling protocol by using the encode/decode methods provided by the pre-builtmarshallers. These encoding and decoding methods provide all necessary basic type transformations in a cross-platform fashion.

Configuration

The xTier™ Marshal Service is configured via the pre-defined **xtier_marshall.xml** configuration file. (A formal specification for this file can be found in the **`\${XTIER_ROOT}/config/dtd/xtier_marshall.dtd file`**.) The configuration file follows the standard xTier™ service configuration pattern. The only configurable property for the marshal service is the marshallable factories. A sample configuration follows:

```
<xtier-marshal>
  <region name="examples">
    <!-- Example factory. -->
    <factory>
      <ioc policy="new">
        <java class="com.fitechlabs.xtier.examples.services.marshal.MarshalFactory"/>
      </ioc>
    </factory>
  </region>
</xtier-marshal>
```

Factories are specified using the <factory> tags and nested IoC objects. (See the xTier™ IoC Service for more details on IoC configuration.) Note that if the developer only uses the pre-defined marshallable containers **MarshalObject** and **MarshalObjectEx** there is no configuration for these factories required (these two containers are handled internally by default). The factories specified in the XML configuration are automatically registering to the service on startup.

Usage

Usage of the marshal service follows the standard pattern of using an xTier™ service: first obtain an instance of the xTier™ kernel that serves as a service registry. Next, once the xTier™ kernel is obtained, it is possible to get an instance of any service, in this case the marshal service. Once the service instance is obtained, the service API can be used. The following example illustrates encoding/decoding (note that exception handling is omitted here for simplicity):

```
// Get the instance of xTier kernel.
XtierKernel xtier = XtierKernel.getInstance();

// Get the instance of 'marshal' service.
MarshalService marshal = xtier.marshal();

// Get the instance of IO marshaller.
IoMarshaller marshaller = marshal.getIoMarshaller();

// Get the destination stream.
OutputStream out = new BufferedOutputStream(new FileOutputStream("test.out"));

// Encoding.
marshaller.encodeFloat64(123.456, out);
marshaller.encodeChar16('S', out);

// Closes stream.
out.close();

// Get the source stream.
InputStream in = new BufferedInputStream(new FileInputStream("test.out"));

// Decoding.
System.out.println("Float64 decoding [" + marshaller.decodeFloat64(in) + "]");
System.out.println("Char16 decoding [" + marshaller.decodeChar16(in) + "]");
```

The following example demonstrates marshalling/de-marshalling:

```
// Get the instance of xTier kernel.
XtierKernel xtier = XtierKernel.getInstance();

// Get the instance of 'marshal' service.
MarshalService marshal = xtier.marshal();
```

```
// Get byte marshaller.
ByteMarshaller marshaller = marshal.getByteMarshaller();

// Creates destination byte array.
byte[] arr = new byte[10000];

// Create & populate user marshallable object
MarshalUserObject userObj = new MarshalUserObject();

// Marshalling.
marshaller.marshallObj(userObj, arr, 0);

// Demarshalling.
MarshalUserObject obj = (MarshalUserObject)marshaller.demarshalObj(arr, 0).getObject();
```

Conclusion

The xTier™ Marshal Service provides two very significant facilities. First, it provides a mechanism for data interchange between the Java and .NET platforms. This is very valuable for building cross platform, cross paradigm, distributed application. The ability to exchange data can be used, for example, in passing input/output parameters in Java/.NET combined applications. Second, the xTier™ Marshal Service provides an extremely fast marshalling/de-marshalling protocol that allows using the service in near real-time systems (nRT). Testing shows that the xTier™ Marshal Service can achieve performance of up to 25x times faster than native Java or .NET serialization. Further, the xTier™ Marshal Service is highly flexible, as developers are not limited to using this built-in marshalling/de-marshalling protocol. The developer is free to implement their own protocols to provide an exact match to their requirements based on the convenient encoding/decoding tools provided by the service. Further the xTier™ Marshal Service provides the facility to perform marshalling using any one of 3 input/output media types – byte array, stream, or nio byte buffer. Developers can use any of these ways according to their needs. All these features and characteristics provide a highly-flexible service and allow the service to be implemented in a wide range of possible applications.

Fitech Laboratories Inc.

Corporate Headquarters

300 Montgomery St., Suite 621
San Francisco, CA 94104
USA

Phone: 1-415-371-8234
Fax: 1-415-371-8237

East Coast Sales Office

330 Madison Ave., 9th floor
New York, NY 10017
USA

Phone: 1-646-495-5076

Fitech Laboratories Japan

Toranomon40 MT Bldg. 3F
5-13-1 Toranomon Minato-ku
Tokyo 105-0001, Japan

Phone: +81-3-5402-7711
Web: www.fitechlabs.co.jp