Transferring large binary data with web services

https://weblogs.java.net/blog/adhirmehta/archive/2010/06/11/transferring-large-binary-data-web-services

Prerequisites

1) Basic understanding about web service
2) Knowledge of base 64 encoding
3) Knowledge of MTOM

Refer resource section for information on these topics.

Introduction

Web service has been evolved from simple request-response mechanism to object oriented style support and now large data transfer. Large data may be of various types like binary, images, database record export in XML or other format etc. Normally, the large data transfer is avoided with the web service however in unavoidable circumstances, if it is required to do; you have few options which will be discussed later in the article.

There is couple of big challenges during the large data transfer through web service

1) How do you make sure that you meet the performance of the application and it should not deteriorate?
2) Memory constraint. Huge files should not be hold in the memory completely.

We'll discuss these challenges later in the article. Now, let's discuss the various data transfer strategies/options to transfer the binary data.

[Note: This article only provides conceptual knowledge.]

Binary data transfer strategies

In the following sections I'll discuss the various data transfer strategies with their advantages and disadvantages.

1) Embedding the binary data in the SOAP envelop

The straight forward option to transfer the binary data is to convert the binary data into base 64 format and embed the base 64 encoding into soap envelop itself. This approach is also called as "by value" approach as the binary data is embedded in the XML document itself. This approach has following advantages and disadvantages.

Advantages:

- Simple to use
- This approach gives applications the ability to process and describe data, based only on the XML component of the data
- It can be positively considered for small binary data transfer.

Drawbacks of embedding binary large data into soap envelop

1) As binary data is part of soap envelop, it requires large memory to hold it.
2) Base64 encoding increases the size of the binary data by a factor of 1.33x of the original size
3) It slows down the application performance.

Here is the API which can be referred/used for implementation

Implementation API:

- Apache commons codec -
- Base64Encoder (From source-code.biz) - http://www.source-code.biz/base64coder/java/

Following sections in the article discuss other approaches also called as "by reference approaches" to overcome these issues. Sending binary data by reference is achieved by attaching pure binary data as external unparsed general entities outside the XML document and then embedding reference URIs to those entities as elements or attribute values. This prevents the unnecessary bloating of data and wasting of processing power.

2) FTP or Network files system

One of the good solutions to transfer the large data via web service is to move out the binary large data from the soap envelop and keep only the reference to binary data as a part of soap response.
To implement this approach, the data can be placed either on FTP server or somewhere on shared location on network and the path to data (file) can be given in the web service request or response.

Following section discuss the advantages and disadvantages of this approach.

Advantages

1) Data is not part of the soap envelop so it does not require huge memory to hold it.
2) Since there is no encoding and decoding requires as in case of base 64, it improves the application performance.
3) You can have separate network line dedicated for FTP which can free up normal application network line resulting in performance improvement.

Disadvantages

1) It requires FTP server.
2) Additional maintenance of FTP server
3) Sometime, may not be best suited for small applications.

Here is the API which can be referred/used for implementation

Implementation API:

- Apache commons - http://commons.apache.org/net/api-release/org/apache/commons/net/ftp/FTPClient.html

3) Message Transmission Optimization Mechanism (MTOM)

MTOM (SOAP Message Transmission Optimization Mechanism) is another specification that focuses on solving the "Attachments" problem. MTOM is actually a "by reference" method. The wire format of a MTOM optimized message is the same as the SOAP with Attachments message. The most noticeable feature of MTOM is the use of the XOP:Include element, to reference the binary attachments (external unparsed general entities) of the message. With the use of this exclusive element, the attached binary content logically becomes inline (by value) with the SOAP document even though it is actually attached separately. This merges the two realms by making it possible to work only with one data model. On a lighter note, MTOM has standardized the referencing mechanism of SwA. The following is an extract from the XOP specification.

At the conceptual level, this binary data can be thought of as being base64-encoded in the XML Document. As this conceptual form might be needed during some processing of the XML document (e.g., for signing the XML document), it is necessary to have a one-to-one correspondence between XML Infosets and XOP Packages. Therefore, the conceptual representation of such binary data is as if it were base64-encoded, using the canonical lexical form of the XML Schema base64Binary datatype. In the reverse direction, XOP is capable of optimizing only base64-encoded Infoset data that is in the canonical lexical form.

The client application sends SOAP Message that contains complex data in Base64Binary encoded format. Base64Binary data type represents arbitrary data (e.g., Images, PDF files, Word Docs) in 65 textual. A sample SOAP Body with Base64Binary encoded element is as follows:

```
 < mtom:ByteEcho>
 < mtom:data>AVBERi0xLjYNJeLjz9MNCjE+DQpzdGFyNCjExOYNCg== mtom:data>
mtom:ByteEcho>
```

An MTOM-aware web services engine detects the presence of *Base64Binary* data, < mtom:*data*> in the example, and makes a decision – to convert the *Base64Binary* data to MIME data with an *XML-binaryOptimization Package* (xop) content type. The data conversion, results in replacing the *Base64Binary*data with an element that references the original raw bytes of the document being transmitted. The raw bytes are appended to the SOAP Message and are separated by a MIME boundary as shown below:

```
  < mtom:ByteEcho>
  < mtom:data> mtom:data>
   mtom:ByteEcho>


--MIMEBoundary000000
content-id: <1.633335845875937500@java.net>
content-type: application/octet-stream
content-transfer-encoding: binary
```

The raw binary data along with the SOAP Message and the MIME Boundary is send over the wire to the Producer. TheProducer then transforms the raw binary data back to *Base64Binary* encoding for other processing. This approach has couple of advantages:

Advantages

1. Effective Transmission: Base64Binary encoded data is ~33% greater than raw byte transmission using MIME. ThereforeMTOM reduces data size by converting Base64Binary encoding to raw bytes for transmission.
2. Data is transferred using streaming approach.

Disadvantages/Constraints

1. HTTP-specific (although it could easily be adapted to other > MIME-based or MIME-like transports)
2. Detecting MTOM messages for dispatch is additional overhead.
3. You may not be able use MTOM if all the parties involved in data transfer do not support MTOM specification.

Here is the API which can be referred/used for implementation

Implementation API:

- Metro - https://metro.dev.java.net/guide/Binary_Attachments__MTOM_.html

Plain HTTP

One of the big genuine question is why just don’t use the http to post and stream the data. It doesn’t require any soap envelop etc. And the answer is yes this can be done provided you are not looking for any web service related benefits.

Conclusion:

I’ve seen different methods of transferring the binary large data through web service however each method has its own advantages and disadvantages. Now, the question is raised which one is better and answer is: depending upon your application environment, you will have to choose one of the options.

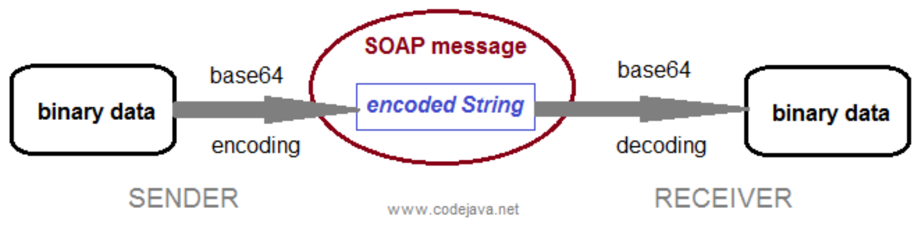
However in my opinion, here is the order of precedence of selection.

1) Use base64 encoding if binary data is very small and it does not substantially impact the performance.
2) Use FTP or network share if binary data is large.
3) Prefer MTOM if all the involved parties in data transfer support the MTOM as it is one of the best way to stream the content.
4) Use plain HTTP if it is just upload/download of the file.

http://www.codejava.net/java-ee/web-services/java-web-services-binary-data-transfer-example-base64-encoding

In the approach employed by this article, the binary data is embedded directly in the SOAP envelop using base64 text encoding. In other words, the raw binary data is converted to an encoded String which is value of an XML element in the SOAP message. Upon receiving the SOAP message, the receiver decodes the encoded String in order to re-construct the original binary data.

The following picture depicts this process:



This approach is the simplest way and is only suitable for transferring a small amount of binary data. It becomes very inefficient when transferring a large amount of binary data because the base64 text encoding technique bloats the data by a factor of 1.33x (UTF-8 text encoding) or 2.66x (UTF-16 text encoding) of the original size. Also, the encoding/decoding process slows down the application performance.

This approach is also often referred as “by value” or “inline attachment” method. Now, let’s go through an example application that is used to transfer small binary files.

NOTE: To optimize the binary data transfer, see: Using MTOM to optimize binary data transfer with JAX-WS web services.

2. Coding the Web Service Endpoint Interface

Let’s define an endpoint interface as follows:

```
package net.codejava.ws.binary.server;

import javax.jws.WebMethod;
import javax.jws.WebService;

/**
 * A web service endpoint interface.
 * @author www.codejava.net
 *
 */
@WebService
public interface FileTransferer {
    @WebMethod
    public void upload(String fileName, byte[] imageBytes);

    @WebMethod
    public byte[] download(String fileName);
}
```

This endpoint defines two web methods, `upload()` and `download()` which allows the client to upload/download a file by sending/receiving a chunk of bytes.

3. Coding Web Service Endpoint Implementation

Write an implementation for the `FileTransferer` interface as follows:

```
package net.codejava.ws.binary.server
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.WebServiceException;

/**
 * A web service implementation of an endpoint interface.
 * @author www.codejava.net
 *
 */
@WebService
public class FileTransfererImpl implements FileTransferer {
    @WebMethod
    public void upload(String fileName, byte[] imageBytes) {

        String filePath = "e:/Test/Server/Upload/" + fileName;

        try {
            FileOutputStream fos = new FileOutputStream(filePath);
            BufferedOutputStream outputStream = new BufferedOutputStream(fos);
            outputStream.write(imageBytes);
            outputStream.close();

            System.out.println("Received file: " + filePath);

        } catch (IOException ex) {
            System.err.println(ex);
            throw new WebServiceException(ex);
        }
    }

    @WebMethod
    public byte[] download(String fileName) {
        String filePath = "e:/Test/Server/Download/" + fileName;
        System.out.println("Sending file: " + filePath);

        try {
            File file = new File(filePath);
            FileInputStream fis = new FileInputStream(file);
            BufferedInputStream inputStream = new BufferedInputStream(fis);
            byte[] fileBytes = new byte[(int) file.length()];
            inputStream.read(fileBytes);
            inputStream.close();

            return fileBytes;
        } catch (IOException ex) {
            System.err.println(ex);
            throw new WebServiceException(ex);
        }
    }
}
```

As we can see, the `upload()` method saves the received bytes array to file specified by the given `fileName`; and the`download()` method reads the requested file and returns its content as a bytes array.

4. Coding the Server Program

Let’s create a server program that publishes the above endpoint implementation as follows:

```
package net.codejava.ws.binary.server;

import javax.xml.ws.Endpoint;

/**
 * A simple web service server.
 * @author www.codejava.net
 *
 */
public class WebServiceServer {

    public static void main(String[] args) {
        String bindingURI = "http://localhost:9898/codejava/fileService";
        FileTransferer service = new FileTransfererImpl();
        Endpoint.publish(bindingURI, service);
        System.out.println("Server started at: " + bindingURI);
    }
}
```

Now run this program and we should see the following message in the console.

```
Server started at: http://localhost:9898/codejava/fileService
```

The server is now waiting for request. Next, we are going to generate the web service client code.

5. Generating Client Code

It’s recommended to use the **`wsimport`** tool in order to generate web services client code in Java. Execute the following command at the command prompt:

```
wsimport -keep -p net.codejava.ws.binary.client
http://localhost:9898/codejava/fileService?wsdl
```

The **`wsimport`** tool generates necessary client code and puts .java source files under the package`net.codejava.ws.binary.client`. Here’s the list of generated files:

-
    - `Download.java`
    - `DownloadResponse.java`
    - `FileTransfererImpl.java`
    - `FileTransfererImplService.java`
    - `ObjectFactory.java`
    - `package-info.java`
    - `Upload.java`
    - `UploadResponse.java`

NOTES: You must ensure that the sever program is running before executing the **`wsimport`** command.

6. Coding the Client Program

Based on the generated web services client code, we can write a client program as follows:

```
package net.codejava.ws.binary.client;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * A client program that connects to a web service in order to upload
 * and download files.
 * @author www.codejava.net
 *
 */
public class WebServiceClient {

    public static void main(String[] args) {
        // connects to the web service
        FileTransfererImplService client = new FileTransfererImplService();
        FileTransfererImpl service = client.getFileTransfererImplPort();

        String fileName = "binary.png";
        String filePath = "e:/Test/Client/Upload/" + fileName;
        File file = new File(filePath);

        // uploads a file
        try {
            FileInputStream fis = new FileInputStream(file);
            BufferedInputStream inputStream = new BufferedInputStream(fis);
            byte[] imageBytes = new byte[(int) file.length()];
            inputStream.read(imageBytes);

            service.upload(file.getName(), imageBytes);

            inputStream.close();
            System.out.println("File uploaded: " + filePath);
        } catch (IOException ex) {
            System.err.println(ex);
        }

        // downloads another file
        fileName = "camera.png";
        filePath = "E:/Test/Client/Download/" + fileName;
        byte[] fileBytes = service.download(fileName);

        try {
            FileOutputStream fos = new FileOutputStream(filePath);
            BufferedOutputStream outputStream = new BufferedOutputStream(fos);
            outputStream.write(fileBytes);
            outputStream.close();

            System.out.println("File downloaded: " + filePath);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

As we can see, this client program connects to the web service then uploads an image file to the service, and finally downloads another image file from the service.

7. Testing the Application

Running the client program we should see the following output:

```
File uploaded: e:/Test/Client/Upload/binary.png
File downloaded: E:/Test/Client/Download/camera.png
```

And here's the server's output:

```
Received file: e:/Test/Server/Upload/binary.png
Sending file: e:/Test/Server/Download/camera.png
```

Using a monitoring tool such as *TCP/IP Monitor* in Eclipse IDE, we can spot the SOAP messages delivered along with the request and response as follows:

- **Invoking the `upload()` method:**
  - Client Request:

```
GET /codejava/fileService?wsdl HTTP/1.1
User-Agent: Java/1.7.0_17
Host: localhost:9898
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

POST /codejava/fileService HTTP/1.1
Accept: text/xml, multipart/related
Content-Type: text/xml; charset=utf-8
SOAPAction:
"http://server.binary.ws.codejava.net/FileTransfererImpl/uploadRequest"
User-Agent: JAX-WS RI 2.2.4-b01
Host: localhost:9898
Connection: keep-alive
Content-Length: 1971

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:upload xmlns:ns2="http://server.binary.ws.codejava.net/">
            <arg0>binary.png</arg0>
            <arg1>iVBORw0KGgoAAAANSUhEUgAAABAAAAAQCAMAAAAoLQ9TAA....</arg1>
        </ns2:upload>
    </S:Body>
</S:Envelope>
```

  -
  - Server Response (WSDL XML is omitted):

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml;charset=utf-8
Date: Fri, 11 Oct 2013 02:48:45 GMT

-- WSDL XML (ommitted for saving space)....


HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8
Date: Fri, 11 Oct 2013 02:48:45 GMT

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:uploadResponse xmlns:ns2="http://server.binary.ws.codejava.net/"/>
    </S:Body>
</S:Envelope>
```

- **Invoking the `download()` method:**
  - Client Request:

```
POST /codejava/fileService HTTP/1.1
Accept: text/xml, multipart/related
Content-Type: text/xml; charset=utf-8
SOAPAction:
"http://server.binary.ws.codejava.net/FileTransfererImpl/downloadRequest"
User-Agent: JAX-WS RI 2.2.4-b01
Host: localhost:9898
Connection: keep-alive
Content-Length: 218

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:download xmlns:ns2="http://server.binary.ws.codejava.net/">
            <arg0>camera.png</arg0>
        </ns2:download>
    </S:Body>
</S:Envelope>
```

  - Server Response:

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8
Date: Fri, 11 Oct 2013 02:48:45 GMT

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:downloadResponse xmlns:ns2="http://server.binary.ws.codejava.net/">
            <return>iVBORw0KGgoAAAANSUhEUgAAABAAA...</return>
        </ns2:downloadResponse>
    </S:Body>
</S:Envelope>
```

CONCLUSION: As we mentioned earlier, this binary data transfer approach is only well-suited for working with small binary data because the base64 text encoding mechanism bloats data size greatly and slows down application performance.

Using MTOM to optimize binary data transfer with JAX-WS web services

http://www.codejava.net/java-ee/web-services/using-mtom-to-optimize-binary-data-transfer-with-jax-ws-web-services

In this Java web services tutorial, we are going to discuss how **MTOM** (*Message Transmission Optimization Mechanism*) can be used to optimize binary data transfer through web services with **JAX-WS** (*Java API for XML-based Web Services*). We will go from background of **MTOM** and its usages to development of a simple web services application that can transfer large binary data (upload and download files) in the optimized way.

1. Why MTOM?

By default, binary data is converted to `base64Binary` or `hexBinary` XML data type within a SOAP envelope, meaning that the raw bytes are encoded as a String using base64 technique. For example, the following XML snippet is content of such a SOAP message:

```
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:upload xmlns:ns2="http://server.mtom.ws.codejava.net/">
            <arg0>binary.png</arg0>
            <arg1>iVBORw0KGgoAAAANSUhEUgAAABAAAAAQCAMAAAAoL...5CYII=</arg1>
        </ns2:upload>
    </S:Body>
</S:Envelope>
```

Look at the text inside the element **`<arg1>`** - it is the encoded form of the binary data in base64 format. Basically, the base64 encoding technique bloats the original data by a factor of 1.33x (with UTF-8 encoding) or 2.66x (with UTF-16), so it becomes very inefficient when dealing with a large amount of data. To overcome this drawback, **MTOM** is defined as a specification for optimizing the transmission of this kind of data type in SOAP messages, and **XOP** (*XML-binary Optimized Packaging*) is the concrete implementation. The following example is content of a HTTP POST request which is generated by **MTOM/XOP**:

```
POST /codejava/fileService HTTP/1.1
Accept: text/xml, multipart/related
Content-Type: multipart/related;start="<rootpart*8b39dc38-7f35-437f-920f-
b99b6b2c9888@example.jaxws.sun.com>";
    type="application/xop+xml";boundary="uuid:8b39dc38-7f35-437f-920f-
b99b6b2c9888";start-info="text/xml"
SOAPAction: "http://server.mtom.ws.codejava.net/FileTransfererImpl/uploadRequest"
User-Agent: JAX-WS RI 2.2.4-b01
Host: localhost:8787
Connection: keep-alive
Content-Length: 2154

--uuid:8b39dc38-7f35-437f-920f-b99b6b2c9888
Content-Id: <rootpart*8b39dc38-7f35-437f-920f-b99b6b2c9888@example.jaxws.sun.com>
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:upload xmlns:ns2="http://server.mtom.ws.codejava.net/">
            <arg0>binary.png</arg0>
            <arg1>
                <xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
                    href="cid:187eff8e-fc5c-4aa5-8a89-
1e09e153ade6@example.jaxws.sun.com">
                </xop:Include>
            </arg1>
        </ns2:upload>
    </S:Body>
</S:Envelope>

--uuid:8b39dc38-7f35-437f-920f-b99b6b2c9888
Content-Id: <187eff8e-fc5c-4aa5-8a89-1e09e153ade6@example.jaxws.sun.com>
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
```

```
[binary octet stream]
```

Here, the request's content type becomes `multipart/related` in which the SOAP message and the binary data are separated as individual parts of a MIME message. The SOAP message itself doesn't contain the binary data. Instead, it has a reference to the part that contains the actual binary data:

```
<xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
    href="cid:187eff8e-fc5c-4aa5-8a89-1e09e153ade6@example.jaxws.sun.com">
</xop:Include>
```

And the binary data is attached to the request as a MIME attachment which is outside the SOAP message.

2. Enabling MTOM in JAX-WS

In JAX-WS, it's easy to enable MTOM for a web service endpoint by using either the **`@MTOM`** or **`@BindingType`** annotations. At the client side, MTOM can be enabled either by passing a new instance of **`MTOMFeature`** class when getting a reference to the web service endpoint (port), or by calling the **`SOAPBinding.setMTOMEnabled(true)`** method on the binding provider object. Here are the usages and examples in details.

Enabling MTOM for the web service endpoint

The following example illustrates a web service endpoint is annotated with the **`@MTOM`** annotation:

```
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.soap.MTOM;

@WebService
@MTOM
public class MyWebService {

    @WebMethod
    public void upload(byte[] data) {
        // implementation details...
    }
}
```

That enables MTOM support for the `MyWebService` endpoint. The **`@MTOM`** annotation has two optional parameters:

- **`enabled`**: specifies whether MTOM feature is enabled (`true`) or disabled (`false`).
- **`threshold`**: specifies the size (in bytes) above which the binary data will be sent as attachment. This would be useful to enable MTOM only for data which is larger than a specified amount.

This example shows MTOM is used but disabled:

```
@WebService
@MTOM(enabled = false)
public class MyWebService {
}
```

This example shows MTOM is enabled only for binary data which is larger than 10KB (10240 bytes):

```
@WebService
@MTOM(threshold = 10240)
public class MyWebService {
}
```

An alternative way is using the **`@BindingType`** annotation with an appropriate value for the SOAP version used. For example:

- Enabling MTOM with SOAP version 1.1:

  ```
  import javax.xml.ws.BindingType;
  import javax.xml.ws.soap.SOAPBinding;

  @WebService
  @BindingType(value = SOAPBinding.SOAP11HTTP_MTOM_BINDING)
  public class MyWebService {
  }
  ```

- Enabling MTOM with SOAP version 1.2:

```
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.SOAPBinding;

@WebService
@BindingType(value = SOAPBinding.SOAP12HTTP_MTOM_BINDING)
public class MyWebService {
}
```

From the above examples, we can see that using the `@MTOM` annotation is preferred as its succinct and flexibility (enabled/disabled and threshold).

Enabling MTOM for the client

The following example shows how to enable MTOM at the client by passing a new instance of the `MTOMFeature` class when getting a proxy reference the web service endpoint:

```
import javax.xml.ws.soap.MTOMFeature;

MyWebServiceService service = new MyWebServiceService();
MyWebService port = service.getMyWebServicePort(new MTOMFeature());
```

Suppose that the `MyWebServiceService` and `MyWebService` classes are generated by the **wsimport** tool. And similar to the`@MTOM` annotation, we can also specify the `enabled` and `threshold` parameters in the `MTOMFeature` class' constructor like this:

```
boolean enabled = true;
int threshold = 10240;
MyWebService port = service.getMyWebServicePort(new MTOMFeature(enabled, threshold));
```

And here's an alternative way, calling the `SOAPBinding.setMTOMEnabled(true)` method:

```
MyWebServiceService service = new MyWebServiceService();
MyWebService port = service.getMyWebServicePort();

BindingProvider provider = (BindingProvider) port;
SOAPBinding soapBinding = (SOAPBinding) provider.getBinding();
soapBinding.setMTOMEnabled(true);
```

So far we have understood the advantages of MTOM and how to apply it for a web service end point and client. Now, let's go through a complete sample application.

3. Coding Web Service Endpoint

The following class is for a web service endpoint implementation that offers two operations for binary data transfer - files upload and download:

```
package net.codejava.ws.mtom.server;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.soap.MTOM;



/**
 * A web service endpoint implementation that demonstrates the usage of
 * MTOM (Message Transmission Optimization Mechanism).
 * @author www.codejava.net
 *
 */
@WebService
@MTOM(enabled = true, threshold = 10240)
public class FileTransferer {
    @WebMethod
    public void upload(String fileName, byte[] imageBytes) {

        String filePath = "e:/Test/Server/Upload/" + fileName;

        try {
            FileOutputStream fos = new FileOutputStream(filePath);
            BufferedOutputStream outputStream = new BufferedOutputStream(fos);
            outputStream.write(imageBytes);
            outputStream.close();

            System.out.println("Received file: " + filePath);

        } catch (IOException ex) {
            System.err.println(ex);
            throw new WebServiceException(ex);
        }
    }

    @WebMethod
    public byte[] download(String fileName) {
        String filePath = "e:/Test/Server/Download/" + fileName;
        System.out.println("Sending file: " + filePath);

        try {
            File file = new File(filePath);
            FileInputStream fis = new FileInputStream(file);
            BufferedInputStream inputStream = new BufferedInputStream(fis);
            byte[] fileBytes = new byte[(int) file.length()];
            inputStream.read(fileBytes);
            inputStream.close();

            return fileBytes;
        } catch (IOException ex) {
            System.err.println(ex);
            throw new WebServiceException(ex);
        }
    }
}
```

As we can see, the `@MTOM` annotation is used to enable binary transfer optimization with the threshold of 10KB.
NOTE: You should prepare a file to be sent to the client and correct the file path according to your environment in the`download()` method.

4. Coding Server Program

Write a simple server program (console application) that publishes the web services with the following code:

```
package net.codejava.ws.mtom.server;

import javax.xml.ws.Endpoint;

/**
 * A simple web service server.
 * @author www.codejava.net
 *
 */
public class WebServiceServer {

    public static void main(String[] args) {
        String bindingURI = "http://localhost:9898/codejava/fileService";
        FileTransferer service = new FileTransferer();
        Endpoint.publish(bindingURI, service);
        System.out.println("Server started at: " + bindingURI);
    }
}
```

Run this program to start the server. We should see the following output in the console:

```
Server started at: http://localhost:9898/codejava/fileService
```

5. Coding Application Client

Use the **wsimport** tool to generate client code for the above web service endpoint (see instructions included in the attached project), then write code for the application client program as follows:

```java
package net.codejava.ws.mtom.client;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import javax.xml.ws.soap.MTOMFeature;

/**
 * A client program that demonstrates how to use MTOM to optimize binary data
 * transfer with JAX-WS web services.
 * @author www.codejava.net
 *
 */
public class WebServiceAppClient {

    public static void main(String[] args) {
        // connects to the web service
        FileTransfererService service = new FileTransfererService();
        FileTransferer port = service.getFileTransfererPort(new MTOMFeature(10240));

        String fileName = "binary.png";
        String filePath = "e:/Test/Client/Upload/" + fileName;
        File file = new File(filePath);

        // uploads a file
        try {
            FileInputStream fis = new FileInputStream(file);
            BufferedInputStream inputStream = new BufferedInputStream(fis);
            byte[] imageBytes = new byte[(int) file.length()];
            inputStream.read(imageBytes);

            port.upload(file.getName(), imageBytes);

            inputStream.close();
            System.out.println("File uploaded: " + filePath);
        } catch (IOException ex) {
            System.err.println(ex);
        }

        // downloads another file
        fileName = "camera.png";
        filePath = "E:/Test/Client/Download/" + fileName;
        byte[] fileBytes = port.download(fileName);

        try {
            FileOutputStream fos = new FileOutputStream(filePath);
            BufferedOutputStream outputStream = new BufferedOutputStream(fos);
            outputStream.write(fileBytes);
            outputStream.close();

            System.out.println("File downloaded: " + filePath);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

As we can see, MTOM is enabled by passing an instance of the `MTOMFeature` class with an integer indicates a

threshold value. This program uploads a file by sending a chunk of bytes to the server, and downloads a file by receiving an array of bytes from the server.
NOTE: You should prepare a file to be uploaded to the server and correct the file path according to your environment.

6. Testing the Application and Inspecting SOAP Messages

Now, execute the application client program to test out the web service. We should see the following output at the client:

```
File uploaded: e:/Test/Client/Upload/binary.png
File downloaded: E:/Test/Client/Download/camera.png
```

And output at the server:

```
Server started at: http://localhost:9898/codejava/fileService
Received file: e:/Test/Server/Upload/binary.png
Sending file: e:/Test/Server/Download/camera.png
```

We inspect the SOAP messages in details by using a monitoring tool such as TCP/IP Monitor in Eclipse IDE. Here's the summary (WSDL request/response is omitted):

- For upload operation:
  - Client Request:

```
POST /codejava/fileService HTTP/1.1
Accept: text/xml, multipart/related
Content-Type: multipart/related;start="<rootpart*0d441e76-7247-40de-a6e5-
d49c9bf4a172@example.jaxws.sun.com>";
        type="application/xop+xml";
        boundary="uuid:0d441e76-7247-40de-a6e5-d49c9bf4a172";start-info="text/xml"
SOAPAction: "http://server.mtom.ws.codejava.net/FileTransferer/uploadRequest"
User-Agent: JAX-WS RI 2.2.4-b01
Host: localhost:9898
Connection: keep-alive
Content-Length: 2133

--uuid:0d441e76-7247-40de-a6e5-d49c9bf4a172
Content-Id: <rootpart*0d441e76-7247-40de-a6e5-d49c9bf4a172@example.jaxws.sun.com>
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:upload xmlns:ns2="http://server.mtom.ws.codejava.net/">
            <arg0>binary.png</arg0>
            <arg1>
                <Include xmlns="http://www.w3.org/2004/08/xop/include"
                    href="cid:876daf6e-de3b-41ff-b8dc-
0d85b3a8951e@example.jaxws.sun.com"/>
            </arg1>
        </ns2:upload>
    </S:Body>
</S:Envelope>

--uuid:0d441e76-7247-40de-a6e5-d49c9bf4a172
Content-Id: <876daf6e-de3b-41ff-b8dc-0d85b3a8951e@example.jaxws.sun.com>
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

[binary octet stream]
```

- Server Response:

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: multipart/related;start="<rootpart*e8611ccc-2442-4d56-8b17-
d15eb6138c24@example.jaxws.sun.com>";
    type="application/xop+xml";
    boundary="uuid:e8611ccc-2442-4d56-8b17-d15eb6138c24";start-info="text/xml"
Date: Fri, 29 Nov 2013 09:57:13 GMT

--uuid:e8611ccc-2442-4d56-8b17-d15eb6138c24
Content-Id: <rootpart*e8611ccc-2442-4d56-8b17-d15eb6138c24@example.jaxws.sun.com>
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:uploadResponse
xmlns:ns2="http://server.mtom.ws.codejava.net/"></ns2:uploadResponse>
    </S:Body>
</S:Envelope>

--uuid:e8611ccc-2442-4d56-8b17-d15eb6138c24--
```

- For download operation:
  - Client Request:

```
POST /codejava/fileService HTTP/1.1
Accept: text/xml, multipart/related
Content-Type: multipart/related;start="<rootpart*95d74f1f-5c21-474b-9481-
400785c18ae5@example.jaxws.sun.com>";
    type="application/xop+xml";
    boundary="uuid:95d74f1f-5c21-474b-9481-400785c18ae5";start-info="text/xml"
SOAPAction: "http://server.mtom.ws.codejava.net/FileTransferer/downloadRequest"
User-Agent: JAX-WS RI 2.2.4-b01
Host: localhost:9898
Connection: keep-alive
Content-Length: 493

--uuid:95d74f1f-5c21-474b-9481-400785c18ae5
Content-Id: <rootpart*95d74f1f-5c21-474b-9481-400785c18ae5@example.jaxws.sun.com>
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:download xmlns:ns2="http://server.mtom.ws.codejava.net/">
            <arg0>camera.png</arg0>
        </ns2:download>
    </S:Body>
</S:Envelope>

--uuid:95d74f1f-5c21-474b-9481-400785c18ae5--
```

- Server Response:

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: multipart/related;start="<rootpart*b581cf1e-507a-4b6a-b1c6-
0bf16e1278c6@example.jaxws.sun.com>";
    type="application/xop+xml";
    boundary="uuid:b581cf1e-507a-4b6a-b1c6-0bf16e1278c6";start-info="text/xml"
Date: Fri, 29 Nov 2013 10:30:12 GMT

--uuid:b581cf1e-507a-4b6a-b1c6-0bf16e1278c6
Content-Id: <rootpart*b581cf1e-507a-4b6a-b1c6-0bf16e1278c6@example.jaxws.sun.com>
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:downloadResponse xmlns:ns2="http://server.mtom.ws.codejava.net/">
            <return>
                <Include xmlns="http://www.w3.org/2004/08/xop/include"
                    href="cid:6a5fab52-948d-44aa-b89e-
072e05af6a60@example.jaxws.sun.com"/>
            </return>
        </ns2:downloadResponse>
    </S:Body>
</S:Envelope>



--uuid:b581cf1e-507a-4b6a-b1c6-0bf16e1278c6
Content-Id: <6a5fab52-948d-44aa-b89e-072e05af6a60@example.jaxws.sun.com>
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

[binary octet stream]
```

So far we have finished discussing what MTOM is and how it is applied in JAX-WS in terms of optimization of binary data transfer via web services.

http://www.informit.com/articles/article.aspx?p=327825&seqNum=17

Using SOAP to Send Binary Data

Our example messages to date have been fairly small, but we can easily imagine wanting to use SOAP to send large binary blobs of data. For example, consider an automated insurance claim registry—remote agents might use SOAP-enabled software to submit new claims to a central server, and part of the data associated with a claim might be digital images recording damages or the environment around an accident. Since XML can't directly encode true 8-bit binary data at present, a simple way to do this kind of thing might be to use the XML Schema type `base64binary` and encode your images as base64 text inside the XML:

```
<soap:Envelope
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soap:Body>
 <submitClaim>
  <accountNumber>5XJ45-3B2</accountNumber>
  <eventType>accident</eventType>
  <image imageType="jpg" xsi:type="base64binary">
   4f3e9b0...(rest of encoded image)
  </image>
 </submitClaim>
 </soap:Body>
</soap:Envelope>
```

This technique works, but it's not particularly efficient in terms of bandwidth, and it takes processing time to encode and decode bytes to and from base64. Email has been using the Multipurpose Internet Mail Extensions (MIME) standard for some time now to do this job, and MIME allows the encoding of 8-bit binary. MIME is also the basis for some of the data encoding used in HTTP; since HTTP software can usually deal with MIME, it might be nice if there were a way to integrate the SOAP protocol with this standard and a more efficient way of sending binary data.

SOAP with Attachments and DIME

In late 2000, HP and Microsoft released a specification called "SOAP Messages with Attachments." The spec describes a simple way to use the multiref encoding in SOAP 1.1 to reference MIME-encoded attachment parts. We won't go into much detail here; if you want to read the spec, you can find it at http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211.
The basic idea behind *SOAP with Attachments (SwA)* is that you use the same HREF trick you saw in the section "Object Graphs" to insert a reference to the data in the SOAP message instead of directly encoding it. In the SwA case, however, you use the content-id (cid) of the MIME part containing the data you're interested in as the reference instead of the ID of some XML. So, the message encoded earlier would look something like this:

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary;
 type=application/soap+xml;start="<claim@insurance.com>"

--MIME_boundary
Content-Type: application/soap+xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim@insurance.com>

<soap:Envelope
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
 <soap:Body>
 <submitClaim>
  <accountNumber>5XJ45-3B2</accountNumber>
  <eventType>accident</eventType>
  <image href="cid:image@insurance.com"/>
 </submitClaim>
 </soap:Body>
</soap:Envelope>

--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <image@insurance.com>

...binary JPG image...

--MIME_boundary--
```

Another technology called *Direct Internet Message Encapsulation (DIME)* , from Microsoft and IBM, used a similar technique, except that the on-the-wire encoding was smaller and more efficient than MIME. DIME was submitted to the IETF in 2002 but has since lost the support of even its original authors.

SwA and DIME are great technologies, and they get the job done, but there are a few problems. The main issue is that both SwA and DIME introduce a data structure that is explicitly outside the realm of the XML data model. In other words, if an intermediary received the earlier MIME message and wanted to digitally sign or encrypt the SOAP body, it would need rules that told it how the content in the MIME attachment was related to the SOAP envelope. Those rules weren't formalized for SwA/DIME. Therefore, tools and software that work with the XML data model need to be modified in order to understand the SwA/DIME packaging structure and have a way to access the data embedded in the MIME attachments.

Various XML and Web service visionaries began discussing the general issue of merging binary content with the XML data model in earnest. As a result, several proposals are now evolving to solve this problem in an architecturally cleaner fashion.

PASWA, MTOM, and XOP

In April 2003, the *"Proposed Infoset Addendum to SOAP With Attachments" (PASWA)* g document was released by several companies including Microsoft, AT&T, and SAP. PASWA introduced a logical model for including binary content directly into a SOAP infoset. Physically, the messages that PASWA deals with look almost identical to our two earlier examples (the image encoded first as base64 inline with the XML and then as a MIME attachment)—the difference is in how we think about the attachments. Instead of thinking of the MIME-encoded image as a separate entity that is explicitly referred to in the SOAP envelope, we logically think of it as if it were still inline with the XML. In other words, the MIME packaging is an optimization, and implementations need to ensure that processors looking at the SOAP data model for purposes of encryption or signing still see the actual data as if it were base64-encoded in the XML.

In July 2003, after a long series of conversations between the XML Protocol Group and the PASWA supporters, the *Message Transmission Optimization Mechanism (MTOM)* g was born, owned by the XMLP group. It reframed the ideas in PASWA into an abstract feature to better sync with the SOAP 1.2 extensibility model, and then offered an implementation of that feature over HTTP. The serialization mechanism is called *XML-Binary Optimized Packaging (XOP)* g; it was factored into a separate spec so that it could also be used in non-SOAP contexts.

As an example, we slightly modified the earlier insurance claim by augmenting the XML with a content-type attribute (from the XOP spec) that tells us what MIME content type to use when serializing this infoset using XOP. Here's the new version:

```
<soap:Envelope
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xop-mime="http://www.w3.org/2003/12/xop/mime">
 <soap:Body>
 <submitClaim>
  <accountNumber>5XJ45-3B2</accountNumber>
  <eventType>accident</eventType>
  <image xop-mime:content-type="image/jpeg"
      xsi:type="base64binary">
   4f3e9b0...(rest of encoded image)
  </image>
 </submitClaim>
 </soap:Body>
</soap:Envelope>
```

An MTOM/XOP version of our modified insurance claim looks like this:

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary;
 type=application/soap+xml;start="<claim@insurance.com>"

--MIME_boundary
Content-Type: application/soap+xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim@insurance.com>

<soap:Envelope
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:xop='http://www.w3.org/2003/12/xop/include'
 xmlns:xop-mime='http://www.w3.org/2003/12/xop/mime'>
 <soap:Body>
 <submitClaim>
  <accountNumber>5XJ45-3B2</accountNumber>
  <eventType>accident</eventType>
  <image xop-mime:content-type='image/jpeg'>
   <xop:Include href="cid:image@insurance.com"/>
  </image>
 </submitClaim>
 </soap:Body>
</soap:Envelope>

--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <image@insurance.com>

...binary JPG image...

--MIME_boundary--
```

Essentially, it's the same on the wire as the SwA version, but it uses the `xop:Include>` element instead of just the `href` attribute. The real difference is architectural, since we imagine tools and APIs will manipulate this message exactly as if it were an XML data model.

MTOM and XOP are on their way to being released by the XML Protocol Working Group some time in 2004, and it remains to be seen how well they will be accepted by the broader user community. Early feedback has been very positive, however, and the authors of this book are behind the idea of a unified data model for XML and binary content.