USER MANUAL

Beam Jive Consulting

Table of contents

1 Introduction

This manual will introduce you to the Offbeat server.

1.1 What is Offbeat server?

Offbeat server is a real-time application and communication server written in Java. The Offbeat server uses a simple and light-weight XML based protocol in messaging.

1.2 Who should use the Offbeat server?

The Offbeat server suits best for the developers who are familiar with the Java programming language and XML. When fast response times and client-to-client communications are required, the Offbeat server is a good choice.

The client-side development can be done in any programming language that supports TCP sockets and XML parsing. For example Flash ActionScript, Java and C++ are suitable languages.

1.3 What can you do with the Offbeat server?

The Offbeat server includes some nice features, such as database connection pooling, object pooling, user variables and groups. Basically it is possible to create any kind of applications with it. You can create database applications, chats, messaging boards, real-time logging and monitoring applications, just to name a few.

1.4 How does it work?

The Offbeat protocol is quite similar to the normal HTTP request-response model. Client applications pass variables in requests to the Offbeat server and it returns an XML document as a response. New applications can be programmed in a very easy fashion and the same application can then be used by any client that supports TCP sockets and XML.

It is possible to create new applications for the server. Applications are loaded into the memory when the server starts. Each application consists of one or more Java class files. Each file can then be called from the client to produce an XML document.

2 Protocol

This chapter will introduce you to the XML based protocol used by the Offbeat server and the client applications.

The Offbeat protocol is unique in the way it processes the requests. The response messages can be sent directly to another user, to groups of users or to everyone using the same application. In the Offbeat server the requests are always processed by using the following pattern:

- Read the request
- Parse the request
- Get the requested application (from an internal list)
- Set user variables (request variables, application variables, group variables…)
- Execute the application (create the XML response message)
- Send the response (to the user who sent the request, to other clients, to groups…)

2.1 Requests

The request messages are always in the same XML format. All request contain the request ID, name of the file that is called and a set of request variables. The basic structure of a request message is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<REQUEST FILE="filename.xma" REQUEST_ID="123123">
   <ITEM NAME="var1"><![CDATA[Value for variable 1]]></ITEM>
   <ITEM NAME="var2"><![CDATA[Value for variable 2]]></ITEM>

   ...

   <ITEM NAME="varN"><![CDATA[Value for variable n]]></ITEM>
</REQUEST>
```

The request ID is an unique number generated by the Offbeat client library. It is used to map the request messages to response messages. All data of the request variables are inside of CDATA blocks. This makes it possible to send for example HTML data in requests (HTML tags will not mess up the XML syntax).

2.2 Responses

A response message is generated every time a request comes in. The response messages always contain a root element MSG that is generated by the server. Everything inside the root elements is generated by the application that was called. The document may contain XML data or character data.

2.2.1 Response messages

The response messages are sent back to the client who sent the request. In the user-written Offbeat application this means that the response document is first generated and then the method response.send() is called. The user will receive an XML document containing the data that was generated in the server application. The format of the response messages is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="metafile.xma" REQUEST_ID="123123" ERRORS="0">

   ... Your data here

</MSG>
```

The TYPE-attribute tells that it is a normal response message. The REQUEST_ID-attribute helps the client library to map the response message to a request (an appropriate callback will be called by the client). The ERRORS-attribute tells the number of errors in the response. If there are errors in the response, the response XML document will look like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="metafile.xma" REQUEST_ID="123123" ERRORS="2">
   <ERROR CODE="0">Error description</ERROR>
   <ERROR CODE="3">Error description</ERROR>
</MSG>
```

It can be seen that the response contains two errors. Error codes make it possible to display the error messages in your own language. **All server generated error codes are bigger than zero, custom errors, generated by you, should always be negative.** Error descriptions contain an English explanation of the error and in some cases also a stack trace of the application. The error codes are listed below:

```
0 = Malformed request
1 = Maximum request size exceeded
2 = Exception while processing the content
3 = Wrong type of file ending
4 = File not found
```

2.2.2 Push messages

Clients can receive push messages from other clients that are connected to the same Offbeat server. The push messages always contain a root element and some other data. The format of a push message is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="1" FILE="myfile.xma" SENDER="clients public ID">

   ... Any data here

</MSG>
```

The TYPE-attribute tells that it is a push message. The SENDER-attribute can be used to map the message to its sender. The client can examine the FILE-attribute to find out where the message came from (and therefore know what to do with the message).

2.2.3 Acknowledgement messages

The acknowledgement messages are sent back to the client when the response.send() method is not called in the Offbeat server application. The client does not need to react to these messages. They are used to make the client API (Application Programming Interface) more simple. The format of the acknowledgement messages is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="2" FILE="myfile.xma" REQUEST_ID="123123"></MSG>
```

When an acknowledgement message arrives to the client, the Offbeat library will check the REQUEST_ID-attribute and remove the corresponding request from the client library's internal request stack.

2.2.4 Disconnect messages

When a client disconnects (or he/she loses connection to the internet), a disconnect message will be sent to all users that are registered to the same application. A client can be registered to more than one application at the same time. The registering and unregistering is done by using the application.register(String name) and application.unregister() methods in an Offbeat application. If both client A and client B have been registered to the application MyApplication and then client A closes the connection, the client B will receive a disconnect message that contains the public ID of the client A. The disconnect messages are always in the following format.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="3" USER_ID="publicID" NAME="users name" />
```

The TYPE-attribute tells that it is a disconnect message. The USER_ID-attribute contains the public ID of the client who has disconnected. The NAME-attribute is an attribute that tells the name of the client that disconnected. It may be used for example in chat-like applications to hold the nick name of a client.

2.2.5 Log messages

Clients may receive log messages. When the user.receiveLogMessages(boolean receive) method has been called in the Offbeat application, the user will receive all log messages that are sent by using the logger.log(String message) method. This feature can be used to create custom monitoring interfaces to the Offbeat applications. The log messages are always in the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="4"><![CDATA[Message data]]></MSG>
```

3 Requests

3.1 General

Client applications can send requests to the Offbeat server. The requests may contain zero or more request variables that can be read in the application that is called. The requests, as well as responses, are \u0000 terminated XML documents (strings). This means that the server reads the incoming data until it reads a \u0000 character.

3.2 Reading a variable from the request

In an Offbeat application the request variables can be read by using the request.getVar(String name) method. The method returns the variable as String if it exists and null if it does not exist in the request. The following code listing illustrates a simple application that echoes a request variable back to the client.

```
import com.bjc.content.*;

public class EchoVariable extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
        response.addData( request.getVar( "myVar" ) );
        response.send();
   }
}
```

3.3 Checking if a variable exists in the request object

Sometimes it is wise to check the existence of a request variable before trying to read it. This can be done by using the request.isSet(String name) method. The method simply returns true or false.

4 Responses

4.1 General

The response messages are generated by the user-written Offbeat applications. The responses are generated by using the response-object. The methods in the response object are used to add XML data to the response message. The response-object contains the following methods:

- void startNode(String name)
- void setAttribute(String name, String value)
- void endNode(String name)
- void addData(String data)
- void addCData(String cdata)
- void addRSNode(String name, ResultSet rs)

- void addGroupList()
- void addUsersInGroup(String gid)
- void addUserList()

- void send()
- void sendUser(String uid)
- void sendGroup(String gid)
- void sendAll()

These methods are used to generate XML documents and to send them back to the client and possibly to other clients as well.

4.2 Adding data to the response message

Normal character data can be added to the response message by using the addData(String data) and addCData(String cdata) methods. The difference between these methods is that the addCData-method adds the text inside of a CDATA block. Inside the CDATA block you can have HTML tags and they will not mess up the XML syntax. The following code listing is a simple example of how to add some character data to the response.

```
import com.bjc.content.*;

public class OutputSomeData extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
        response.addCData( "Just some data" );
        response.send();
   }
}
```

The response message would look like the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="OutputSomeData.xma" REQUEST_ID="12" ERRORS="0">
<![CDATA[Just some data]]>
</MSG>
```

The methods startNode, setAttribute and endNode can be used to create XML elements to the response. In the following example these methods are used to create an XML response that describes a person.

```
import com.bjc.content.*;

public class GetPerson extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
        // Set the first name
        response.startNode( "FIRSTNAME" );
        response.addData( "John" );
        response.endNode( "FIRSTNAME" );

        // Set the last name
        response.startNode( "LASTNAME" );
        response.addData( "Doe" );
        response.endNode( "LASTNAME" );

        // Send to the client
        response.send();
   }
}
```

This example application would generate the following output.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="GetPerson.xma" REQUEST_ID="14" ERRORS="0">
   <FIRSTNAME>John</FIRSTNAME>
   <LASTNAME>Doe</LASTNAME>
</MSG>
```

4.3 Adding a ResultSet to the response

It is also possible to add a Java ResultSet object to the response by using the response.addRSNode(String name, ResultSet rs). This method simplifies the code because there is no need to iterate through the resultset. The following example application shows how the Java ResultSets can be added to the response. In the example we get all persons from the database and add the results to the response.

```
import com.bjc.content.*;
import java.sql.*;

public class GetPersons extends MetaContent implements MetaAccess
{
   public void processContent()throws CustomException
   {
        // Run the query by using the connection 'con1'
        ResultSet rs = database.runQuery("con1","SELECT * FROM person");

        // Add the query results to the response
        response.addRSNode( "PERSONS", rs );

        // Send to the client
        response.send();
   }
}
```

If there are fields id, firstname and lastname in the person table, this application would generate a response message like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="GetPersons.xma" REQUEST_ID="41" ERRORS="0">
   <PERSONS>
          <ROW>
                <COL NAME="id"><![CDATA[13]]></COL>
                <COL NAME="firstname"><![CDATA[John]]></COL>
                <COL NAME="lastname"><![CDATA[Doe]]></COL>
          </ROW>

          ... More persons ...

   </PERSONS>
</MSG>
```

4.4 Adding a list of existing groups to the response

With the method response.addGroupList() it is possible to add a list of all groups in the application to the response message. The following example demonstrates how the method should be used.

```
import com.bjc.content.*;

public class GetGroups extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
        // Add group list to the response
        response.addGroupList();
        // Send to the client
        response.send();
   }
}
```

If there are two groups in the application, the response message would look like the following XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="GetGroups.xma" REQUEST_ID="50" ERRORS="0">
   <GROUP ID=”a65d76ad65f65f65451b3123” NAME=”My Group1” USERS=”75”>
         The description of the group
   </GROUP>
   <GROUP ID=”cd786c778a8745c7aad35d54” NAME=”My Group2” USERS=”4”>
         The description of the group number 2
   </GROUP>
</MSG>
```

The ID-attribute is the unique group ID of the group. It is an ID that the Offbeat server generates when it creates the group. The group ID is used to access the group (ie. Send message to a group).

NAME-attribute is the human-readable name of the group. Inside of the GROUP tags there is the human-readable description of the group. The programmer defines the group name and the description when the group is being created.

USERS-attribute tells how many users are there in the group at the moment.

4.5 Adding a list of users in a group to the response

The method response.addUsersInGroup(String gid) can be used to add a list of members in a group to the response message. The parameter gid is the unique group ID of the group. The following code listing shows how the method should be used.

```
import com.bjc.content.*;

public class GetGroupMembers extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
        // Add a list of members in the group to the response
        response.addUsersInGroup( request.getVar( “theGID” ) );
        // Send to the client
        response.send();
   }
}
```

In the example, a request variable ‘theGID’ is read and passed to the addUsersInGroup method. If there are two members in the group, the response message will take the following form.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="GetGroupMembers.xma" REQUEST_ID="72" ERRORS="0">
   <USER ID=”8dhfd125a12fg3g216”>The users name in the group</USER>
   <USER ID=”51fs15g554g23a3dd3”>Users name in the group</USER>
</MSG>
```

The ID-attribute is the public ID of the client. The public ID is used to access a certain user (ie. To send a message to the user).

4.6 Adding a list of users in the application to the response

Some times it is useful to know all users who are registered to the same application. This can be achieved by using the response.addUserList() method. A call to this method will add a list of all users to the response message. All users who have registered to the application by calling the application.register(String name) method will be on the list. The following example shows how theresponse.addUserList() method should be used.

```
import com.bjc.content.*;

public class GetAppMembers extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
        // Add a list of members in the group to the response
        response.addUsersList();
        // Send to the client
        response.send();
   }
}
```

The output would look like the XML document below.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="GetAppMembers.xma" REQUEST_ID="76" ERRORS="0">
   <USER ID=”8dhfd125a12fg3g216”>Users name in the application</USER>
   <USER ID=”51fs15g554g23a3dd3”>Users name in the application</USER>
</MSG>
```

Again, the ID is the public ID of the user. Inside the USER tag, there is the name of the user (given in the application.register method call).

4.7 Controlling the response output

The response message can be sent back to the user who sent the request, to another user, to a group or to all users in the same application. The following example illustrates how a message can be sent to another user (basic chat messaging).

```
import com.bjc.content.*;

public class SendMessage extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
        // Add a list of members in the group to the response
        response.addCData( request.getVar( “theMsg” ) );

        // Send private message to another client
        response.sendUser( request.getVar( ”theClient” ) );
   }
}
```

The user who sent the request, would receive an acknowledgement message and the other user would receive a push message. In the same manner, the message can be sent to groups and to all registered users in the application.

5 Applications

5.1 General

The Offbeat server is based on the idea of applications. It is possible to create new applications as needed. However, **it is also possible to access more than one application through one socket connection**. The application object is used to access the properties of the application and to manage the groups inside the application.

5.2 Accessing applications

The scope of the application object is always the application in which the requested file belongs to. In other words, you can not access ApplicationA properties from the ApplicationB.

5.3 Groups

There are four methods in the application-object that can be used to handle groups in the given application. The methods are:

- String createGroup(String name, String description)
- boolean removeGroup(String gid)
- boolean groupExists(String gid)
- Vector getGroups()

These methods can be used to create and remove groups dynamically. The following code listing shows how to create a group dynamically.

```
import com.bjc.content.*;

public class CreateGroup extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
        String groupID = new String();

        // Create the group
        groupID = application.createGroup( request.getVar( "name" ),
request.getVar( "desc" ) );

        // Add the ID of the created group to the response
        response.addData( groupID );

        // Send the response back to the client
        response.send();
   }
}
```

5.4 Application properties

It is possible to add properties to an application. The application variables are shared with all clients. The following methods of the application object can be used to manage the properties:

- void setProperty(String name, String value)
- String getProperty(String name)
- void unsetProperty(String name)

Please notice that the application properties can not be locked. If two or more clients read the existing value of a property and then set it to a new value based on the old value, it is possible that new value is not what it is supposed to be. This happens because the clients read the old value before setting the new value.

5.5 Registering and unregistering users

It is possible to register user with an application by calling the application.register() method. The registered users will receive the client disconnect messages and all the messages that are sent by calling the response.sendAll() method. The application.unregister() method just removes the registration from the memory.

5.6 Controlling the ban list

There are three methods in the application-object that can be used to control the ban list:

- void removeBan(String ip)
- void ban(String publicID)
- void banIP(String ip)

These methods are pretty self-descriptive. The application.removeBan method removes the ban for the given IP address. With the application.ban and application.banIP you can dynamically ban access to the server.

6 Groups

6.1 General

It is possible to create groups to applications dynamically by using the methods in the application object or statically by using a XML configuration file groups.xml (in the groups folder of the application). Groups may have properties and an unlimited amount of users as members.

6.2 Group ID

The Offbeat server gives a unique group ID to each group when they are created. The group ID is used to access the groups.

6.3 Predefined groups

The groups can be defined in a configuration file or by method calls at the run time. The predefined groups can be defined in a file called groups.xml. The file should locate in the applications sub directory called conf. If there is no need for predefined groups, the file does not have to be present. The syntax of the groups.xml is as follows:

```
<GROUPS>
   <GROUP>
         <NAME>Name of the group 1</NAME>
         <DESCRIPTION>The description of the group 1</DESCRIPTION>
   </GROUP>
   <GROUP>
         <NAME>Name of the group 2</NAME>
         <DESCRIPTION>The description of the group 2</DESCRIPTION>
   </GROUP>

   ... more groups here

</GROUPS>
```

6.4 Adding user to a group

Method group.addUserToGroup(String gid, String uid, String name) can be used to add user to a group. The parameter gid is the unique ID of the group. The parameter uid is the public ID of the user that should be added to the group. The parameter name is the name of the user in that group. The following code listing shows how a user can be added to a group.

```
import com.bjc.content.*;

public class JoinGroup extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
       group.addUserToGroup( request.getVar( "myGID" ),
user.getPublicID(), request.getVar( "myName" ) );
   }
}
```

In the example listing, the user.getPublicID() method is used to get the public ID of the user who made the request.

6.5 Testing user existence in a group

To test if a user has joined a group, the method group.isInGroup(String gid, String uid) can be called. The first parameter is the group ID and the second parameter is the users public ID.

6.6 Removing user from a group

Method group.removeUserFromGroup(String gid, String uid) can be used to remove user from a group.

6.7 Getting the user count in a group

Sometimes it is useful to know how many users have joined a group. This can be done by using the group.getUserCount(String gid) method which returns the user count as an integer number.

6.8 Group names and descriptions

The name and description of a group can be obtained by using methods group.getGroupName(String gid) and group.getGroupDescription(String gid). These methods return the information as a String.

6.9 Group properties

By using the group properties it is possible to share data amongst the users in the group and maintain some additional information about the group. The properties can be set, get and unset by using the following methods of the group object:

- void setProperty(String gid, String name, String value)
- String getProperty(String gid, String name)
- void unsetProperty(String gid, String name)

7 Users

7.1 General

Users are the clients that are connected to the Offbeat server. It is possible to set user properties by using the methods in the user object. User has also unique ID's (unique inside the whole server) that can be used in various ways.

User object also contains a method receiveLogMessages(boolean receive). If it is called with true parameter, the user will start receiving all log messages generated by a call to logger.log(String msg) inside the application.

7.2 Public ID and private ID

The public ID should be used for the messaging purposes. It is the ID that can be exposed to other users. The private ID is also unique and similar to the public ID. However, it should be used privately so that the other clients do not know it. Methods user.getPublicID() and user.privateID() can be used to get the IDs. Both methods return a String.

7.3 User properties

The following methods of the user object can be used to manage the user properties:

- void setProperty(String name, String value)
- String getProperty(String name)
- void unsetProperty(String name)

The properties can contain any information. If the client is using more than one application, the properties are available only in application in which it was set. The user properties are private; other clients have no access to them.

7.4 Dropping and banning a user

Sometimes it may be necessary to drop or ban one single user. It can be done by calling the user.ban() or user.drop() method.

8 Database connections

8.1 General

The Offbeat server offers an easy and efficient way to pool database connections and to execute database queries. The database connections are created when the server starts and pooled in the memory. This feature makes the database queries easy and fast. Instead of creating a new connection for each request, the server gets the connections from the pool. The server uses the standard JDBC database API to connect to the databases.

It possible to specify the size of the connection pool (how many connections are initially made to the database). The connection pool works so that if there are no free connections left in the pool, the server will create a new one. The administrator should adjust the size of the pool based on the expected load. If the database server does not allow more connections to be made, the queries will fail and a null value will be returned instead of a ResultSet object.

The connection pool also takes care of the timed-out connections by reconnecting to the database.

8.2 Creating a connection pool

The connection pool can be created by adding an XML configuration file called db_connections.xml to the conf directory of the application. The following listing shows the format of the connection pool configuration file (connections to a MySQL server).

```
<DB_CONNECTIONS>
   <CONNECTION NAME="conn1" POOLSIZE="15">
          <DRIVER>com.mysql.jdbc.Driver</DRIVER>
          <DB_URL>jdbc:mysql://127.0.0.1/my_db1</DB_URL>
          <USERNAME>my_username</USERNAME>
          <PASSWORD>my_password</PASSWORD>
   </CONNECTION>
   <CONNECTION NAME="conn2" POOLSIZE="20">
          <DRIVER>com.mysql.jdbc.Driver</DRIVER>
          <DB_URL>jdbc:mysql://10.0.0.2/my_db2</DB_URL>
          <USERNAME>my_username</USERNAME>
          <PASSWORD>my_password</PASSWORD>
   </CONNECTION>
</DB_CONNECTIONS>
```

There are two different connections that will be pooled in the server. The NAME-attribute of the CONNECTION tag is used in the application code to specify the database connection. The POOLSIZE-attribute tells the server how many connections it should be pooling. The DRIVER tag is the name of the JDBC driver. The DB_URL specifies the database server URL and the name of the database. Tags USERNAME and PASSWORD are the username and password to the database specified in the DB_URL tag.

Notice that the configuration file parameters follow the same syntax as is used when connecting to a database normally through the JDBC API.

The correct **JDBC driver has to be present and correctly set to the PATH environment variable**.

8.3 Running queries

When the connection pool has been configured correctly, the connections may be used through the database object in the Offbeat applications. The database object has method runQuery(String conn, String SQL) that can be used to run queries that return a ResultSet (SELECT statements). The first parameter is the connection name (specified in the db_connections.xml) and the second parameter is the SQL statement that will be executed.

Together with the response.addRSNode(String name, ResultSet rs) method, it is easy to run queries. The following example listing shows how the queries can be executed.

```
import com.bjc.content.*;
import java.sql.ResultSet;

public class GetNews extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
      // Try to run the query
      ResultSet rs = database.runQuery( "con", "SELECT * FROM news" );

      if( rs != null )
      {
        // Add the results to the response
        response.addRsNode( "NEWS", rs );
      }
      else
      {
        // Query failed - raise exception
        throw new CustomException( -1, "Unable to run query: con" );
      }

      // Send the response back to user
      response.send();
   }
}
```

This is the recommended way to run queries. For testing purposes it might be sometimes enough to write just:

```
import com.bjc.content.*;

public class GetNews extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
     response.addRsNode( "NEWS", database.runQuery( "con", "SELECT *
FROM news" ) );
      response.send();
   }
}
```

8.4 Running updates

It is possible to run update queries to the database by using the database.runUpdate(String con, String SQL) method. This method should be used to run INSERT, UPDATE and DELETE

777 statements. The method returns the number of affected rows as an int or a negative number if the
778 query fails. The error return codes are:

779 -1: Unknown error in the query

780 -2: The connection was not found

781 -3: Unable to connect to the database

9 Object pooling

9.1 General

Object pool makes it possible to store any Java objects in the memory. Objects may be database connections, Strings, byte arrays etc. There is one object pool per one user. An object in the pool is accessible only by the user who put the object there; the objects are not shared resources. Objects in the pool exist only in the application where they were set.

9.2 Setting object to the pool

Objects can be set to the pool by calling the setObject(String name, Object o) method of the pool object. The following code listing shows how the method is used.

```
import com.bjc.content.*;

public class StoreData extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
     Long obj = new Long( 123 );
     pool.setObject( "myObject", obj );
   }
}
```

9.3 Reading an object from the pool

Objects can be read from the pool by calling the getObject(String name) method of the pool object. The method has Object as the return type, so it is necessary to cast the object to the type of your object. The following code listing shows how to read a Long object from the pool.

```
import com.bjc.content.*;

public class GetStoredData extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
     Long obj = (Long) pool.getObject( "myObject" );
     ... processing goes on ...
   }
}
```

9.4 Checking if the object exists in the pool

Sometimes it is useful to check the existence of an object in pool. This can be done by calling the isSet(String name) method of the pool object. The method simply return a boolean value that indicates the existence of the object.

9.5 Removing an object from the pool

The method unsetObject(String name) of the pool object can be used to remove an object from the pool. If there is an object with the given name, it will be removed. Otherwise nothing will happen.

10 Logging

10.1 General

Logging in the Offbeat server applications can be done by using the built-in logger object and its log(String msg) method. The log method produces log messages that have an automatic timestamp. The timestamp is formatted in the dd.mm.yyyy hh:mm:ss format.

A call to the log method will cause the log message to be written to the log file of the application. The log file resides in the log directory of the application and is called application.log.

The same log messages may be automatically sent to connected clients.

The main log file of the server is called internal.log and is located in the logs directory of the server. All client connects, disconnects etc. will be written to the man log file.

10.2 Generating log messages

The log messages can be created by calling the log method of the logger object. The following code listing shows how it is done.

```
import com.bjc.content.*;

public class MyFoo extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
      ... do something ...

      logger.log( ”We are doing something here!” );

      ... do something ...
   }
}
```

10.3 Receiving log messages

Any client can receive the log messages. This can be done by calling user.receiveLogMessages(true) in the application.

11 Error handing

11.1 General

In some cases it is necessary to raise an exception. For example, if a database query fails, it is wise to generate an exception with a descriptive error message and code.

11.2 Generating an exception

The error situations can be handled cleanly by throwing a CustomException with a descriptive message and error code. The following code listing shows how it can be done.

```
import com.bjc.content.*;

public class MyFoo2 extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
      ... do something ...

      // Something is going wrong -raise an exception
     throw new CustomException( -1, "My description" );

      ... do something ...
   }
}
```

11.3 Handling errors on the client

Only normal response messages can contain errors. Push messages are not sent if an exception is thrown while processing the content. In the response messages, there is always the ERRORS-attribute present. If the ERRORS-attribute is greater than zero, there are errors in the response. The following response message has two errors.

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="metafile.xma" REQUEST_ID="123123" ERRORS="2">
   <ERROR CODE="-1">Error description 1</ERROR>
   <ERROR CODE="-2">Error description 2</ERROR>
</MSG>
```

12 Installing the server

The installation of the Offbeat server is quite straight forward. To install the Offbeat server on a Windows machine, double click the Offbeat-vX_X_Xp.exe (where X is the version number) in the package. To install the Java version, unzip the Offbeat-vX_X_Xp-VERSION-JAVA.zip to the directory of your choice.

To develop Offbeat applications, path to the bin directory of the server should be set to the CLASSPATH environment variable of the computer (or the user who starts the server).

13 Configuring and starting the server

The server does not need a lot of configuration. The only main configuration file, server.xml, is in the conf directory under the Offbeat root directory. The following listing shows the required format of the server.xml file.

```
<?xml version="1.0" encoding="utf-8"?>
<SERVER>
   <PROPERTY NAME="xml_port">8384</PROPERTY>
   <PROPERTY NAME="client_timeout">0</PROPERTY>
   <PROPERTY NAME="max_clients">800</PROPERTY>
   <PROPERTY NAME="max_request_length">1000000</PROPERTY>
   <PROPERTY NAME="max_response_length">10000000</PROPERTY>
   <PROPERTY NAME="class_loading_mode">0</PROPERTY>
</SERVER>
```

The server properties are named so that they are easy to understand. The xml_port property is the port number to which the server listens to. The client_timeout property is the time in milliseconds that the server waits an idle connection before closing it. The client_timeout value 0 means that there is no timeout. The max_clients property is the maximum number of concurrent client connections. The max_request_length property defines the maximum size of the request messages in bytes. The max_response_length property defines the maximum size of the response messages in bytes. The class_loading_mode can be either 0 or 1. The default value is 0 and it means that all clients have their own copies of the classes. When the class_loading_mode is 1, it means that all clients load the classes from the same place when needed.

The server can be started by going to the bin directory of the Offbeat server and typing:

```
java -jar Offbeat-v1_0_0p.jar
```

in the DOS command prompt or in a Linux shell. If the server does not start, please check the Java version by typing:

```
java -version
```

Version should be 1.4.X. In many performance tests, the server edition of the JRE has been faster than the default client edition. To use the server mode, start the Offbeat server by typing:

```
java -jar -server Offbeat-v1_0_0p.jar
```

You can also use all available switches in your JVM to make the server more robust. For example, you could use the Xmx and Xms switches to control the JVM memory consumption. In the following example, the JVM will allocate 128 MB of memory from the heap at the start and the maximum heap memory consumption is set to 256 MB:

```
java -jar -server -Xmx256m -Xms128 Offbeat-v1_0_0p.jar
```

14 Creating an application

14.1 General

This chapter will show you how to create an application with the Offbeat server.

14.2 The directory structure

All Offbeat applications are located under the apps directory. It is possible to create new applications just by creating a new subdirectory under the apps directory. Every Offbeat application should have the following directories:

- conf
- logs
- meta_classes

The conf directory contains two configuration files: db_connections.xml and groups.xml. Both files are optional. The logs directory contains the application log file called application.log. The meta_classes directory contains the application classes. The meta_classes directory can have also subdirectories.

14.3 Writing the first application

The application files should be located in the meta_classes directory. As you have probably noticed, the Offbeat applications may be constructed of one or more Java class files. The skeleton of an Offbeat application file is shown in the code listing below.

```
import com.bjc.content.*;

public class NameOfTheClass extends MetaContent implements MetaAccess
{
   public void processContent() throws CustomException
   {
      // The code comes here
   }
}
```

The name of the class should always match with the file name. For example, if you declare a class called MyClass, you should save the file as MyClass.java and compile it to MyClass.class.

It is possible to include any external java classes and libraries. The classes should be in the CLASSPATH of the user who starts the server. It is a good idea to divide your external code into packages.

The java files may also be in the meta_classes directory because the Offbeat server only loads files that have .class file extension. A better practice would still be to keep the source files in another directory under the application root. For example, /OFFBEAT/apps/MyApp/source, is a good choice.

14.4 Compiling the application

To compile the application, you need to have the Offbeat bin directory in the CLASSPATH environment variable (this is also true when running the server). The easiest way is to move to the application's meta_classes directory and type java MyClass.java. This will then create a class file and the server may be started.

If the Offbeat bin directory is not in the CLASSPATH, you can compile all java files in your meta_classes directory by typing:

```
javac -classpath .;..\..\bin *.java
```

on Windows, or

```
javac -classpath .;../../bin *.java
```

on Linux.

14.5 How the applications are loaded

The Offbeat server uses a custom class loader that loads every class into a b-tree structure at the start time. Every class is then in the memory of the server, and therefore fast to access when a request comes in.

Currently there is no mechanism to load, unload or reload classes at the runtime. The feature might be implemented in the future, if there is a need for it.

14.6 How to disable an application

Sometimes you may need to disable an application temporarily. This can be done by changing the application name so that the word 'Inactive_' preceeds the application name. For example, the application MyApp would become Inactive_MyApp.

Notice that this does not remove the applications from the server memory, it affects so that the application will not be loaded when the server is started.

14.7 Flash cross-domain policy files

The Offbeat server can serve cross-domain policy files for Flash applications. The cross-domain policy files are needed when the swf document is loaded from different domain than where the Offbeat server is running or the Flash client connects to Offbeat port under 1024.

The Offbeat server loads the crossdomain.xml from conf directory and returns it to Flash client. For example, if you have the server listening to port 80 on host 127.0.0.1, you could use the following code in Flash to allow the connection:

```
System.security.loadPolicyFile( "xmlsocket://127.0.0.1:80" );
```

This should work on Flash Players version 7,0,19,0 and later. Please see www.macromedia.com for more details.

15 Hardware requirements

The Offbeat server can run on a mid-range PC computer. To get better performance, the speed of the CPU and the amount memory, are the key factors. Faster CPU means faster response times and more memory means more concurrent clients. For a good performance, at least 256 MB RAM is needed.

16 Software requirements

16.1 Operating system

The Offbeat server has been developed in pure Java, so it should run on all platforms that have a Java runtime environment available. The server has been tested on Windows 2000, Windows XP, Windows NT and Red Hat Linux 8, 9 and 10. We do not promise the correct functioning on other operating systems and versions.

The Native Windows binary version runs on most Windows machines. It was tested on Windows XP, Windows 2000 and Windows NT.

The native Linux version runs on all major Linux versions.

16.2 Java platform

The J2SE 1.4.X is required to compile applications for the Offbeat server. Java runtime environment (JRE) 1.4.x is enough to run the Java version of the server.

17 Server characteristics

17.1 Number of concurrent users

The number of concurrent users is limited by the computer CPU speed, amount of available RAM and the maximum allowed amount of threads by the operating system. The server was load tested on a 1 GHz CPU machine with 128 MB memory. Then the maximum number of concurrent users was about 700 when the clients were actually sending requests to the server.

In one of the load tests, we connected 7000 clients to the Offbeat server which was running on a 1.5 GHz Celeron with 512 MB RAM memory. In this case the client activity was very low, so the processor had time to accept more new clients.

With a high speed machine with a lot of memory, it is possible to get a lot more concurrent connections.

For the most demanding applications, like real-time controlling (very CPU intensive), the maximum number of concurrent users may be significantly lower.

17.2 Size of the requests

The size of the requests is not very important issue in the Offbeat server. If the request contains a lot of request variables, parsing of the request takes naturally longer.

17.3 Size of the response messages

Because Offbeat server does a lot of in-memory processing, the size of the response messages is important. It better to request a small amount of data many times than to request everything at the same time. The round-trip times are very small, so it is a good idea to get only the data that can be displayed at one time.

17.4 Response times

During the test runs, the fastest round-trip times were about 1 ms. Database queries and other bigger operations take naturally more time than the simple operations.

18 Programming API

This chapter describes the programming API of the Offbeat server. There are currently eight objects that can be used to control the server output.

18.1 The application object

The application object can be used to set application wide settings. Each application has one application object. The application object cannot be shared between the applications.

String createGroup(String name, String description)
This method can be used to create new group to an application **Parameters:** name: A human-readable name for the group description: Description of the group **Returns:** An unique group ID for the group as String
boolean removeGroup(String gid)
This method can be used to remove a group from an application **Parameters:** gid: The group ID of the group that should be removed **Returns:** True if the group existed, false if the group was not found
boolean groupExists(String gid)
This method can be used to check if a group exists in the application **Parameters:** gid: The group ID **Returns:** True if the group exists, false if it does not
Vector getGroups()
Get a list of all groups in the application. **Parameters:** **Returns:** A vector that contains the group IDs of all groups in the application

void setProperty(String name, String value)
Set an application property. The properties are shared with all users of the application. **Parameters:** name: Name of the property value: Value of the property **Returns:**
String getProperty(String name)
Get an application property. **Parameters:** name: The name of the property to get **Returns:** The value of the property, or null if the property does not exist
void unsetProperty(String name)
Remove an application property. **Parameters:** name: Name of the property to remove **Returns:**
void register(String name)
Register the user with the application. When this is called, the user will receive the disconnect messages and all messages that are sent by using the sendAll() method. **Parameters:** name: Name of the user in the application. **Returns:**
void unregister()
Unregister the user. When called, the user will not receive the disconnect messages or the messages sent by calling the sendAll() method. **Parameters:** **Returns:**
String getPath()
Return an absolute path to the application. **Parameters:**

Returns: Path to the application directory
void removeBan(String ip)
Removes the given IP address from the list of banned addresses. **Parameters:** ip: The IP address that should be removed from the ban list **Returns:**
void ban(String publicID)
Can be used to ban a currently connected client. **Parameters:** publicID: Public ID of the client to ban **Returns:**
void banIP(String ip)
Add the given IP to the list of banned addresses **Parameters:** ip: IP address that should be banned **Returns:**
Hashtable getUsers()
Returns a list of users that have registered with the application. The method returns a Hashtable that contains publicID and name for each user. **Parameters:** **Returns:** Hashtable containing all users in the application

18.2 The group object

The group object can be used to control the groups in an application. The group ID is used to access the groups. The application.createGroup(String name, String description) method returns the group ID (gid in the method parameters).

boolean addUserToGroup(String gid, String uid, String name)
This method can be used to add a user to a group.

Parameters:

gid: The group ID of the group where the user should be added to

uid: The public ID of the user that will be added to the group

name: Name of the user in the group

Returns:

True if the user was successfully added to the group, false otherwise

boolean isInGroup(String gid, String uid)

This method can be used to check if a user has already joined a group.

Parameters:

gid: The group ID

uid: Public ID of the user

Returns:

True if the user is in the group, false if not

boolean removeUserFromGroup(String gid, String uid)

This method can be used to remove a user from a group

Parameters:

gid: The group ID

uid: The public ID of the user

Returns:

True if the user was removed, false if the user was not in the group

int getUserCount(String gid)

Get the number of members in a group

Parameters:

gid: The group ID

Returns:

Number of members in the group

void setProperty(String name, String value)

Set a property to the group

Parameters:

name: Name of the property to set

value: Value of the property to set

Returns:

Nothing

String getProperty(String name)

Get a group property **Parameters:** name: Name of the property to get **Returns:** Value of the property or null if the property does not exist
void unsetProperty(String name)
Remove a property from the memory **Parameters:** name: Name of the property to remove **Returns:** Nothing
String getGroupName(String gid)
Get the human-readable name of a group **Parameters:** gid: The group ID **Returns:** The name of the group or null if the group does not exist
String getGroupDescription(String gid)
Get the description of a group **Parameters:** gid: The group ID **Returns:** The description or null if the group does not exist

18.3 The user object

The user object can be used to control the properties of the client who did the request.

String getPrivateID()
Get the private ID of the user **Parameters:** None **Returns:**

The private ID

String getPublicID()

Get the public ID of the user

Parameters:

None

Returns:

The public ID

void setProperty(String name, String value)

Set a property that can be used only by the user (who sent the request). The property stays in the memory until removed with a call to the unsetProperty method or until the user disconnects.

Parameters:

name: Name of the property to set

value: Value of the property to set

Returns:

Nothing

String getProperty(String name)

Get a user property

Parameters:

name: Name of the user property to set

Returns:

Value of the property or null if the property does not exist

void unsetProperty(String name)

Remove a property from the server memory

Parameters:

name: Name of the property to remove

Returns:

Nothing

void receiveLogMessages(boolean receive)

When called with true parameter, the user will receive all log messages in the application.

Parameters:

receive: When true, the user will receive log messages. When false, the user will not receive the log messages.

Returns:

Nothing

void drop()
Closes the connection for the given user. **Parameters:** **Returns:** Nothing
void ban()
Adds the users to the list of banned users (uses the IP address of the user) and closes the connection. **Parameters:** None **Returns:** Nothing
String getIP()
Returns the IP address of the user. **Parameters:** None **Returns:** IP address of the user
String getHost()
Returns the hostname of the user. **Parameters:** None **Returns:** Host name of the user

1084

1085 18.4 The request object

1086 The request object can be used to read the request variables.

1087

String getVar(String name)
Get the value of a request variable **Parameters:** name: Name of the request variable to get

Returns: Value of the request variable or null if the variable is not set
boolean isSet(String name)
This method can be used to test if a request variable exists **Parameters:** name: Name of the request variable **Returns:** True if the request variable is set, false otherwise

18.5 The response object

The response object is used to create the response message and to control the output.

void startNode(String name)
Starts a new XML node. **Parameters:** name: Name of the XML node that is being created **Returns:** Nothing
void setAttribute(String name, String value)
Add an attribute to the lastly added node. Notice that the startNode method should be called before this. This method will not add an attribute to the root node of the response message. **Parameters:** name: Name of the attribute to set value: Value of the attribute **Returns:** Nothing
void endNode(String name)
Create an end node to the response message. For example, the method call response.endNode(“MyNode”) would add </MyNode> to the response message. **Parameters:** name: Name of the end node **Returns:** Nothing
void addData(String data)

Add character data to the response message.

Parameters:

data: Any character data

Returns:

Nothing

void addCData(String data)

Add character data to the response message. Wraps the data inside of a CDATA block. This can be used for example when HTML data has to be added to the response. The XML parsers will not try to parse the CDATA block, therefore the XML document remains valid.

Parameters:

data: Any character data

Returns:

Nothing

void addRSNode(String name, ResultSet rs)

Converts the Java ResultSet object to XML and appends it to the response. The resulting XML data will look like the following:

```
<NAME>
   <ROW>
      <COL NAME="mycol"><![CDATA[yourvalue]]></COL>
   </ROW>
</NAME>
```

Where the NAME tag is what you pass to the method. The attribute NAME of the COL tag is the column name. The actual field data is placed between the COL tags and a CDATA section.

Parameters:

name: Name of the result set

rs: A ResultSet object that contains query results

Returns:

Nothing

void addGroupList()

Adds a list of groups to the response message. The resulting XML will look like the following (there are two groups in this case):

```
<GROUP ID="a65d76ad65f65f65451b3123" NAME="My Group1" USERS="4">
   The description of the group
</GROUP>
<GROUP ID="cd786c778a8745c7aad35d54" NAME="My Group2" USERS="53">
   The description of the group number 2
</GROUP>
```

Parameters:

None

Returns:

Nothing

void addUsersInGroup(String gid)

Adds a list of users in the group to the response message. The group is defined by the parameter gid.

Parameters:

gid: Group ID

Returns:

Nothing

void addUserList()

Add a list of all users in the application. This method lists all users who have registered to the application by using the application.register() method.

Parameters:

None

Returns:

Nothing

void send()

A call to the method will signal the server that the generated XML response message should be sent back to the user who sent the request. It does not matter at which point the method is called, the response is always sent when all code has been executed.

Parameters:

None

Returns:

Nothing

void sendUser(String uid)

A call to this method will cause the server to send the XML response message to another user as a push message. The user is identified by the uid parameter, which is the public ID of a user.

Parameters:

uid: The public ID of the user that the message will be sent

Returns:

Nothing

void sendGroup(String gid)

The generated XML response message will be sent to all users in the group that is defined by

the gid parameter. The message will not be sent to the sender even if he/she is part of the group. **Parameters:** gid: The group ID **Returns:** Nothing
void sendAll()
Send the XML response message to all users in the application. The message will be sent to those users who have registered with the application by using the application.register() method. The message will not be sent to the sender. **Parameters:** None **Returns:** Nothing

18.6 The logger object

The logger object can be used to efficiently write log messages to the applications log file (located in the logs directory of the application). A normal file access is very slow. The log method uses a threaded logger.

void log(String msg)
Append the message to the log file of the application. This method produces timestamp automatically. The timestamp is in dd.mm.yyyy hh:mm:ss format. **Parameters:** msg: Any log message **Returns:** Nothing

18.7 The database object

The database object can be used to execute database queries and updates.

ResultSet runQuery(String con, String sql)
Executes a SELECT query defined by the sql parameter. The query uses the pooled database connections that can be defined in a configuration file. **Parameters:**

con: A valid connection name in the db_connection.xml file sql: An SQL SELECT statement **Returns:** ResultSet object containing the query results, or null if the query failed
void runUpdate(String con, String sql)
Executes an update operation (UPDATE, INSERT or DELETE) to the database. **Parameters:** con: A valid connection name in the db_connection.xml file sql: An SQL statement **Returns:** If the update operation succeeds, the method returns the number of rows modified. If the operation fails, the method returns one of the following negative error codes: -1: Unknown error in the query -2: The connection was not found -3: Unable to connect to the database
Connection getConnection(String name)
Returns a java.sql.Connection object from the database connection pool. The connections are automatically returned to the pool once the processContent method has been executed. **Parameters:** name: Name of the connection (specified in the db_connections.xml) **Returns:** Open connection to a database or null if the connection is not available

18.8 The pool object

The pool object can be used to store any Java objects in the server memory. The objects are stored until removed with the unsetObject method or until the user disconnects. Every user has one object pool and it cannot be shared between the applications or other users. The object pooling is more efficient than serializing the objects to the hard drive.

void setObject(String name, Object o)
Adds new object to the pool. The object may be any Java object, for example Long or ResultSet. **Parameters:** name: Name of the object to set o: Object to set **Returns:** Nothing

Object getObject(String name)

Get an object from the object pool. You may need to cast the object to the type of your object. For example, if you have added an object of the type Long to the object pool, you need to cast the object like in the following:

```
Long myLong = (Long) pool.getObject( "myLongInThePool" );
```

Parameters:

name: Name of the object to get

Returns:

The object, or null if the object does not exist.

boolean isSet(String name)

Check the existence of an object in the object pool.

Parameters:

name: The name of the object to check

Returns:

True if the object exists in the pool, false otherwise.

void unsetObject(String name)

Removes an object from the pool.

Parameters:

name: Name of the object to remove

Returns:

Nothing

Version history

Version	Date	Author	Description
1.0	18.12.2003	Kai Hannonen	First version
1.1	21.4.2004	Kai Hannonen	Fixed typos, changed the response.addGroupList method description.
1.2	2.8.2004	Kai Hannonen	Added new descriptions for new methods.
1.3	10.9.2004	Kai Hannonen	Added references to the Java Offbeat client
1.4	20.9.2004	Kai Hannonen	Added a chapter about the Flash cross-domain policy files