REFERENCE MANUAL

Version 1.1.0

OREGANO

MULTIUSER SERVER

WWW.OREGANO-SERVER.ORG

REFERENCE MANUAL

Version 1.1.0

written by Jens Halm

Table of Contents

CHAPTER 1
Feature Overview

Oregano Server is a Multiuser Server for Flash clients. It is free and released as Open Source under the GNU Lesser General Public License (LGPL). Since it is written in pure Java it runs on many operating systems including Linux, Windows or Mac OS X.

It is useful for client-server applications that require low latency such as real-time chat systems or shared whiteboard applications that would be difficult to develop with traditional HTTP-based solutions.

Groups and User Management

Oregano Server comes with database driven user management and a versatile mechanism for creating and configuring groups. All the interaction takes place in groups. You can create an unlimited number of groups and extend their built-in functionality with server extensions written in Java. Built-in features include messaging or synchronization of the properties of all members of a group. Simple multiuser applications like chat systems can be written with a few lines of ActionScript and without the need to develop server-side extensions.

Real-time Messaging and Serialization

Easily exchange data between clients or push data from the server to the client. Or use persistent messages, so that a user, who is currently offline, receives the message the next time she logs in. The content of messages is not restricted to strings, you can exchange many different ActionScript datatypes and objects, including Date, Array or the generic Object, which will be serialized and converted to their corresponding Java datatypes automatically. No more cumbersome and error-prone parsing of strings.

Synchronization

Properties associated with a user or a group can be modified and synchronized easily with client-side or server-side methods. Synchronization and persistence settings can be configured separately for each property associated with a group or a user. A property can be configured to be synchronized with every client, to be loaded into the server-side cache only or to be read from the database each time it is accessed. Furthermore it can be configured to be persistent or transient. All the typical requirements for synchronization and persistence will be handled automatically, so you can concentrate on the application logic.

Database Connectivity

Oregano Server uses JDBC for database access, so every database that comes with a JDBC driver can be integrated. All SQL statements the server uses are included in an external XML configuration file, so you can easily make adjustments if you want to use proprietary syntax of the database you use. In addition you can create your own tables and custom SQL statements. Simply include them in the XML configuration file and use them from within the ActionScript database objects of the Oregano Client without writing a single line of server-side code!

Logging and Statistics

The server comes with an extensive logging system, which is divided into several categories whose log-level can be adjusted separately in the configuration file. Statistical recordings include the number of logins per day, the maximum number of concurrent connections per day and the average login time per user per day.

Clustering

When the server load gets too high you can distribute Oregano Server to more than one machine. An unlimited number of secondary servers can be integrated. Adding a few lines to configuration files is all you have to do to build a cluster.

Extensibility

You can easily extend Oregano Server through the built-in extension interface. You can extend the functionality of a single group or all the groups in the server, and you can write custom code to be executed before a user is logged in, so you are able to authenticate her with legacy systems. The extensions are plain Java classes that implement one of the five interfaces that serve as a hook into the server.

Documentation

Oregano Server comes with a 150-page user manual. It contains a description of each method of the Client API and Server API, and each attribute of each node of the XML configuration files, as well as example code for a simple chat application that you can use as a starting point for your own application.

CHAPTER 2
Installation

Installing the Java Runtime Environment

To run Oregano Server, you need the JRE 1.4.0 or later. If you are not sure which version is installed on your machine open a shell window and execute the command `java -version`.

You can download the latest JRE from `java.sun.com`. Simply follow the installation instructions on the download page. After you are done installing the JRE you must add the bin directory inside the installation directory to the `PATH` environment variable. The procedure differs for each platform:

Linux (bash)

Add a line such as the following to the end of your ~/.bashrc:

```
export PATH=/path-to-jre/bin:$PATH
```

Linux (C shell)

Add a line such as the following to the end of your ~/.cshrc:

```
set path=(/path-to-jre/bin $path)
```

Windows XP/2000/NT 4

Start the control panel, select System, then Environment. Add the bin directory to the User Variable named PATH, using a semicolon to separate the new entry, like this:

```
c:\path-to-jre/bin;other stuff
```

Windows 98/ME

Add a line such as the following to the end of your AUTOEXEC.BAT:

```
SET PATH=c:\path-to-jre\bin;%PATH%
```

To test whether you did it right, execute the command `java -version` again.

Installing the JDBC driver

Assuming that your database engine is installed and running, you need to install a JDBC driver for that database. For MySQL you can download the driver from `www.mysql.com`. Move the driver to the `jre/lib/ext` directory inside your JRE installation directory. If booting Oregano Server fails with a message like this

```
error initializing connection pool
server boot aborted. see bootlog.txt for details
```

and bootlog.txt contains lines like this

```
creating ConnectionPool
error initializing connection pool
java.lang.ClassNotFoundException: org.gjt.mm.mysql.Driver
    at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
    ...
```

then the JDBC driver is not properly installed.

Another common cause for errors is that MySQL is configured with the skip-networking option. If you cannot connect to MySQL with Oregano although the driver is loaded correctly check that this option is commented out in my.cfg.

Installing Oregano Server

Two archives in the download section at `www.oregano-server.org` are mandatory for developing Flash multiuser applications: The Oregano Server Runtime and the Oregano Client. All other pieces are optional. Download the Oregano Server Runtime archive to your server machine and simply unzip it. Before you can boot Oregano Server, you must create the required tables in your database and edit the start script and configuration file, which will be explained in the next three sections.

The Oregano Client archive contains the ActionScript files required to be included with your Flash movies. Unzip the archive and copy the whole `oregano_as` directory to the directory of your Flash files. Read the introductory sections in the Client API chapter for further instructions.

The documentation for Oregano Server includes the User Manual you are reading right now and javadoc generated documentation in HTML format. The latter contains the same information as the Reference section of the Server API chapter in this manual.

The optional archives in the download section include the Java source code, the simple example application which is described in Chapter 3 and a small Flash admin client.

Creating the Oregano tables

Before you boot Oregano Server for the first time, the required tables must be created from the SQL batch file oregano.sql in your Oregano server runtime directory. For MySQL the following steps are required:

Start the mysql command line tool.

Create a new database with

```
CREATE DATABASE anyName
```

Tell MySQL to use that new database

```
USE anyName
```

Execute the SQL batch file with the command

```
SOURCE /path-to-oregano/oregano.sql
```

If you want to use the Flash Admin Client, you have to insert at least one administrator into the admin table. You can do this with the following SQL statement:

```
INSERT INTO admins VALUES (1,'userID','password','mail@home.com',
    9,NOW(),NOW(),1);
```

Editing config.xml

This is the main configuration file for the Oregano Server. It is included in the config directory inside the server runtime directory. You can leave most of the settings unchanged, as the default values provided in the downloaded file will be sufficient to boot the server. You can edit them later when you start developing your first application.

There are only a few entries that you have to edit before you can successfully boot the server:

Edit the address attribute of the server node inside the serverConfig node. The address can be an IP address or the human-readable name (ie, www.myserver.com).

Edit the following four attributes in the database node:

url

The URL of the database. The format of the URL is specific to the JDBC implementation of your database. For MySQL it looks like this:

```
jdbc:mysql://localhost/anyName?auto-reconnect=true
```

This URL says to use the mysql driver to connect to the database anyName on localhost. For MySQL make sure to include the auto-reconnect flag so that the Oregano connection pool does not have to cope with closed connections. You should find the URL format for your driver in the documentation that comes with it.

driver

The fully classified class name of your driver (e.g. org.gjt.mm.mysql.Driver). Make sure that the Driver is located in the lib/ext directory of your JRE.

userID

The user ID that should be used to connect to the database.

password

The password that should be used to connect to the database.

Editing the start script

If you run Oregano Server on Linux you have to edit start.sh. For Windows the start script is named start.bat. Both are included in the Oregano server runtime directory. On Linux you have to set the file permissions to make it executable:

```
chmod 770 start.sh
```

Open the script in a text editor and add values for the six variables:

SERVER_ADDRESS

The address of the server. Must match the address attribute of the corresponding `server` node in config.xml.

SERVER_TYPE

Can take one of the following values:

Standalone	The server runs on a single machine.
Primary	This server is the primary server of a cluster. Only one machine can act as the primary server.
Secondary	This server is one of an unlimited number of secondary servers in a cluster.

If you want to use the config.xml that was included in the Oregano download, the value must be `Standalone`.

PRIMARY_ADDRESS

The address of the primary server. This variable is only required in the start script of a secondary server, because these machines must register with the primary server during startup. On a Primary or Standalone server leave this entry empty.

RMI_PORT

A port number between 1024 and 65535, corresponding to the rmi attribute of the `ports` node in config.xml. This value is only required in the start script of a secondary server, because these machines must register with the primary server during startup. On a Primary or Standalone server leave this entry empty.

MEMORY_INIT

The initial heap size. Default is `64M`.

MEMORY_MAX

The maximum heap size. Default is `128M`.

Booting Oregano Server

Open a shell window. Change into the Oregano Server runtime directory. Run `start.sh` (on Linux) or `start.bat` (on Windows). On Linux the output is redirected to nohup.out, so you have to peek into that file to know if the server has booted successfully (the best way is to execute `tail -f nohup.out`). When you see the line `server running...` Oregano Server has finished booting and is accepting connections. You can now try the example application that you can download from `www.oregano-server.org`. See Chapter 3 for a description of that application. When the output includes the line

```
server boot aborted. See bootLog.txt for details
```

open that file in the log directory to see what went wrong.

CHAPTER 3
Example Application

If you skip through this manual you might be overwhelmed with all the configuration options and the number of objects in the API. The small example application that you can download from `www.oregano-server.org` might serve as an easy to grasp starting point for your own applications.

Installing and running the application

1. Download the example application from `www.oregano-server.org` and unzip it.
2. Move config.xml from the config directory of the server runtime directory to a safe place, so that you can put it back later.
3. Open config.xml included with the example application download. This configuration file is prepared for use with the example. You only need to add the information that is specific to your machine.
4. Set the `address` attribute of the server node inside the `serverConfig` node. The address can be an IP address or the human-readable name (ie, `www.myserver.com`).
5. Edit the following four attributes in the database node: `url`, `driver`, `userID`, `password`. If you are not sure how to do this, see the Configuration chapter for details.
6. Save the file and copy it to the `config` directory of your server runtime directory.
7. Reboot Oregano Server.
8. Open `login.as` in the `example_as` directory and edit the first line. Set the second parameter of the `init` method to the server address that you specified in config.xml (or to `localhost` if client and server run on the same machine).
9. Open example.fla and start it with "Test Movie" from the Control menu.
10. Click on "new user", choose an avatar and on the next screen choose a nick name and a password and click "OK".
11. Enter one of the four "rooms" by clicking on the small map in the upper right corner.
12. You can try out one of the three features of this application:
    - enter text into the field at the bottom and press return to send the message.
    - move your avatar to a new position by clicking on the stage.
    - change into another room by clicking on the small map in the upper right corner.

This is all you can do with this application. You might want to start a second client to see how it works with multiple connected users. Please note that this is a very simple application that was developed to show how the core pieces of the Client API can be used to build a simple graphical chat application. It does not, for example, handle any kind of depth sorting for the avatars in the rooms, because these kind of features are not related to the Oregano API.

Examining the objects

You can have a look at the source code of the example application in the include files in the `example_as` directory. The most important parts will be explained in the next section:

Login

The login object with methods for login and registration is included in `login.as`.

The first line initializes the Session object which is one of the singletons that are created automatically in the `org.omus` namespace:

```
org.omus.session.init("example","localhost",1655);
```

The login object registers itself with the Session object in its constructor:

```
org.omus.session.addListener(this);
```

The login object handles all four events that the Session object can generate. The `onLogin` and `onRegister` event handlers both call the `init` method where a bunch of listeners are registered with the corresponding objects, because many of the objects available in the `org.omus` namespace will not be created until the user has successfully logged in:

```
LoginObj.prototype.init = function () {
    org.omus.group.addListener(_root.room);
    org.omus.info.addListener(_root.room);
    org.omus.messenger.addListener(_root.chat);
    org.omus.info.loadGroupList();
    popup.hide();
    _root.gotoAndStop(4);
}
```

In addition the onRegister event sets the avatar property to the string identifier for the avatar that the user had chosen:

```
LoginObj.prototype.onRegister = function (extraData) {
    this.init();
    var ps = org.omus.user.getProperties();
    ps.setValue("avatar",this.chosenAv);
    ps.synchronize();
}
```

The configuration for the avatar property in config.xml looks like this:

```
<userProp
    name="avatar"
    type="string"
    persistence="writeOnChange"
    cacheLevel="synchronizeGroup"
/>
```

Since the attribute persistence is set to `writeOnChange`, the new value is automatically stored in the database through the command `ps.synchronize()`. The attribute `cacheLevel` is set to `synchronizeGroup`, so that the value is loaded into all the clients of the current group automatically, because the other clients need to know the value to show the corresponding avatar on the stage.

The actual connect function is quite simple. In case of a registered user it calls:

```
org.omus.session.login(username,password);
```

For a new user the following method is called:

```
org.omus.session.register(username,password,"");
```

Since this application does not require that an email address is specified, the third parameter is an empty string.

The `onLogout` and `onError` event handlers do only create the appropiate popup message.

Room

The Room object with methods for changing groups and moving avatars is included in room.as. When you click on the map in the upper right corner of the stage, the avatar will move into another room.

Since the room object was registered as a listener with the `org.omus.group` object (in the `init` method of the login object) it contains methods that handle the events that the group object can generate.

The `onChange` event handler is invoked when the user has joined another group:

```
RoomObj.prototype.onChange = function (extraData) {
    // remove users of the old room
    for (each in this.users) {
        this.users[each].remove();
    }
    // create all users of the new room
    this.users = new Object();
    var ar = org.omus.group.getAllUserProperties();
    for (var i = 0; i < ar.length; i++) {
        var propSet = ar[i];
        var av = new _root.AvatarObj(propSet,this.depth++);
        this.users[propSet.getOwner()] = av
    }
```

```
        // show graphics for the new room
        _root.main.gotoAndStop(org.omus.group.getName());
        // reload the group list to refresh the numbers on the map
        org.omus.info.loadGroupList();
    }
```

It removes all the avatars of the old group, and creates a new avatar object for each PropertySet object in the array returned by the `getAllUserProperties` method of the group object (each user of the group is represented through her corresponding PropertySet object). The movie clips to display the avatars will be attached in the constructor for the avatar object (see the section on `avatar.as` for details).

If a new user enters the room (i.e. a user controlled by a different client), the `onUserJoined` event handler will be called:

```
RoomObj.prototype.onUserJoined = function (propSet) {
    var av = new _root.AvatarObj(propSet,this.depth++);
    this.users[propSet.getOwner()] = av
    org.omus.info.loadGroupList();
}
```

It creates a new avatar object the same way the `onChange` event handler does. The `onUserLeft` event handler removes the corresponding avatar from the screen.

The `onClick` handler is called when a user clicks on the map in the upper right corner or when she clicks on the main stage. In the latter case the position property of the user is changed and synchronized, so that all other clients get notified. If the main stage was clicked, a random start position for the new room is calculated and then the `change` method of `org.omus.group` is called:

```
RoomObj.prototype.onClick = function (instName) {
    if (instName == "main") {
        // click on main stage -> move avatar
        if (org.omus.group.getName() == "login") return;
        var x = _root.main._xmouse;
        var y = _root.main._ymouse;
        x = Math.max(x,41);
        x = Math.min(x,375);
        y = Math.max(y,155);
        y = Math.min(y,315);
        var ps = org.omus.user.getProperties();
        ps.setValue("position",{x:x,y:y});
        ps.synchronize();
    } else {
        // click on map -> change into new room
        if (instName == org.omus.group.getName()) return;
        var x = parseInt(Math.random()*this.xRange + 41);
        var y = parseInt(Math.random()*this.yRange + 155);
        var ps = org.omus.user.getProperties();
        ps.setValue("position",{x:x,y:y});
        org.omus.group.change(instName,null,true);
    }
}
```

And finally, when the group list was loaded into the client, the corresponding event handler updates the four numbers in the map in the upper right corner:

```
RoomObj.prototype.onLoadGroupList = function (groups) {
    var l = groups.size();
    for (var i = 0; i < l; i++) {
        var row = groups.getRow(i);
        var cnt = row.userCount;
        if (row.name != "login") map["num_" + row.name] = cnt;
    }
}
```

This event handler gets called because the room object was registered as a listener with the `org.omus.info` object in the `init` method of the login object.

Avatar

The Avatar object is included in avatar.as. It has two properties: A reference to its avatar MovieClip and the PropertySet object that belongs to the user. This is the constructor:

```
AvatarObj = function (propSet,dp) {
    this.propSet = propSet;
    this.propSet.addListener(this);
    var username = propSet.getOwner();
    _root.main.attachMovie("Avatar",username,dp);
    this.mc = _root.main[username];
    this.mc.gotoAndStop(propSet.getValue("avatar"));
    this.changePos();
}
```

The avatar clip is attached to the main clip in _root. After that the new clip is sent to the frame that contains the avatar graphics the user had chosen and finally `this.changePos()` is called to move the clip to the right position.

In the second line of the constructor the object registers itself as a listener with the PropertySet object. It handles the `onSynchronize` event that occurs when a property value has changed. Since there are only two properties and the avatar property is never changed, it looks for a property named "position" and moves the avatar to the corresponding position:

```
AvatarObj.prototype.onSynchronize = function (propSet) {
    if (propSet.contains("position")) this.changePos();
}

AvatarObj.prototype.changePos = function () {
    var pos = this.propSet.getValue("position");
    this.mc._x = pos.x;
    this.mc._y = pos.y;
}
```

Chat

The chat object is included in chat.as. It is really simple. It was registered as a listener with the messenger object in the `init` method of the login object. It listens for incoming messages that have the subject `"chat message"`. It adds the incoming messages to the chat array and updates the field on the stage:

```
ChatObj.prototype.onMessage = function (msg) {
    if (msg.getSubject() == "chat message") {
        if (this.txt.length >= 5) this.txt.shift();
        var attach = msg.getAttachment();
        this.txt.push(msg.getSender() + ": " + attach.txt);
        var str = "";
        for (var i = 0; i < this.txt.length; i++) {
            str += this.txt[i] + "\n";
        }
        _root.chatClip.chatText = str;
    }
}
```

When the user hits the return key, the `sendMsg` method is called:

```
ChatObj.prototype.sendMsg = function () {
    if (Key.getCode() == Key.ENTER) {
        var txt = _root.chatClip.input;
        var msg = new org.omus.Message("chat message");
        msg.getAttachment().txt = txt;
        org.omus.messenger.sendToGroup(msg);
        _root.chatClip.input = "";
    }
}
```

The chat object uses Message objects to exchange data whereas the room object uses properties to exchange the avatar position coordinates between clients. See the Exchanging Data section in the Client API chapter for a thorough discussion on when to use which approach.

CHAPTER 4
Configuration

config.xml - the main configuration file

The main configuration file is included in the config directory of your Oregano server runtime directory. This chapter describes each attribute of each node in the file, optional attributes are marked with an *.

If you build a cluster, you only have to edit the configuration file on the Primary Server, all the other servers will load the file from that location.

<login>

Contains three required attributes and no child nodes.

Example:

```
<login
    mode="public"
    versionID="tr-%$4s"
    emailUnique="true"
/>
```

mode

This attribute accepts one of the following two values:

public — every registered user who is not banned is permitted to login

debug — only administrators (users with a permission value set to 9 or 10) are permitted to login

versionID

An arbitrary string. The versionID must be passed to the `init` method of the client-side Session object. If the versionID of a client does not match the value of this attribute, login will be prevented. This way you can make sure that only clients with the current version of your application can log in.

emailUnique

Boolean value. If set to `true` the server prevents that two users with the same email address are registered with the server.

<ports>

Contains one required attribute, three optional attributes and no child nodes.

Example:

```
<ports
     login="1442"
     reconnect="1443"
     admin="1705"
     rmi="1099"
/>
```

login

The login port. A number between 1024 and 65535. This port number must be passed to the `init` method of the client-side Session object.

reconnect *

The reconnect port. A number between 1024 and 65535. This attribute is optional. It must be specified if the Oregano Server runs on more than one machine. This port number must be passed to the `init` method of the client-side Session object.

admin *

The admin port. A number between 1024 and 65535. This attribute is optional. It must be specified if you enable the admin client - see the section on the admin node for details. The port number must be entered on the login screen in the Flash admin client.

rmi *

The RMI port. A number between 1024 and 65535. This attribute is optional. It must be specified if the Oregano Server runs on more than one machine. Note that this port does not have to be open to the public. It will only be used to exchange data between server machines.

<extensions>

Contains four optional attributes and no child nodes. Oregano Server is fully functional without any extensions, so if the built-in functionality is sufficient for your application, you do not need to develop server-side extensions.

Example:

```
<extensions
     keyGenerator="MyGenerator"
     passwordDecoder="MyDecoder"
     loginExtension="Validator"
     logoutExtension="Stats"
/>
```

keyGenerator *

The name of the class to be used to generate keys for password encryption. This attribute is optional. The specified class must implement `org.omus.ext.KeyGenerator` and must always be used in conjunction with a password decoder. See the section on KeyGenerator in the Server API chapter.

passwordDecoder *

The name of the class to be used to decode passwords. This attribute is optional. The specified class must implement `org.omus.ext.PasswordDecoder` and must always be used in conjunction with a key generator. See the section on PasswordDecoder in the Server API chapter.

loginExtension *

The name of the class to be used to extend the login functionality. This attribute is optional. The specified class must implement `org.omus.ext.LoginExtension`. See the section on LoginExtension in the Server API chapter.

logoutExtension *

The name of the class to be used to extend the logout functionality. This attribute is optional. The specified class must implement `org.omus.ext.LogoutExtension`. See the section on LoginExtension in the Server API chapter.

<admin>

Contains one required attribute, three optional attributes and no child nodes. If you do not want to use the Flash admin client for security concerns, set enableClient to `false` and omit the remaining attributes. Note that in Version 1.0.0 there is no alternative. If you want to logout a user or shutdown the server without loss of data, you have to use the Flash admin client. Later versions will come with a command line tool that has the same functionality and can be executed on the server.

Example:

```
<admin
     enableClient="true"
     versionID="myx5§2b"
     keyGenerator="AdminKey"
     passwordDecoder="AdminDecoder"
/>
```

enableClient

Boolean value. Set this attribute to `true` if you want to use the Flash admin client to logout users or shutdown the server.

versionID *

An arbitrary string. The versionID must be entered on the login screen in the Flash admin client. If the versionID of a client does not match the value of this attribute, login will be prevented. This attribute is optional. It must be specified if `enableClient` is set to `true`.

keyGenerator *

The name of the class to be used to generate keys for password encoding. This attribute is optional. The specified class must implement `org.omus.ext.KeyGenerator` and must always be used in conjunction with a password decoder. See the section on KeyGenerator in the Server API chapter.

passwordDecoder *

The name of the class to be used to decode passwords. This attribute is optional. The specified class must implement `org.omus.ext.PasswordDecoder` and must always be used in conjunction with a key generator. See the section on PasswordDecoder in the Server API chapter.

Note that you must implement the corresponding encoder function in the client. Create a new empty Flash file and on the first frame define a function called `encode`:

```
function encode (password, serverKey) {
    var str = ... // encode the password using the serverKey
    return str;
}
```

Export that movie with the name `encode.swf` and move that file to the directory of your Oregano Admin Client.

Before you login with the Admin Client make sure to check "use external password encoder" on the first screen.

<serverConfig>

Contains no attributes and one or more server child nodes. Each machine in use must be configured with a corresponding server node. In most cases Oregano Server will run on one machine, but you can easily build scalable server clusters. See the Installation chapter for more details.

Example:

```
<serverConfig>
    <server
        id="serv1"
        address="www1.myserver.com"
        userLimit="200"
        dynamicGroups="true"
    />
```

```
        <server
            id="serv2"
            address="www2.myserver.com"
            userLimit="200"
            dynamicGroups="true"
        />
    </serverConfig>
```

<server>

Contains four required attributes and no child nodes.

Example:

```
        <server
            id="serv1"
            address="www.myserver.com"
            userLimit="200"
            dynamicGroups="true"
        />
```

id

The id of the server. An arbitrary string that can be used in the groupConfig node, where you must specify the server for each group that you want to be created during startup.

address

The address of the server. Must match the name that you specified in the start script. See the Installation chapter for more details.

userLimit

The maximum number of connections permitted for this server. If Oregano Server runs on more than one machine, each can have its own user limit.

dynamicGroups

Boolean value. If this attribute is set to `true` this machine will be included in the built-in server load balancing, that always chooses the server with the highest difference between the user limit and the number of currently connected users to dynamically create a new group. If this attribute is set to `false`, no groups other than those that were created during startup can be created on this server.

<database>

Contains six required attributes and one child node.

Example:

```
<database
    url="jdbc:mysql://localhost/oregano_db"
    driver="org.gjt.mm.mysql.Driver"
    userID="herbert"
    password="flyn4/re"
    connectionLimit="25"
    transactions="false"
    >
    <typeMap>
        <!-- see typeMap section -->
    </typeMap>
/>
```


url

The URL of the database. The format of the URL is specific to the JDBC implementation of your database. For MySQL it looks like this:

```
jdbc:mysql://localhost/omus10
```

This URL says to use the mysql driver to connect to the database `omus10` on `localhost`. You should find the URL format for your driver in the documentation that comes with it.

driver

The fully classified class name of your driver (e.g. `org.gjt.mm.mysql.Driver`). Make sure that the Driver is included in the `jre/lib/ext` directory of your `JRE` installation.

userID

The user ID that should be used to connect to the database.

password

The password that should be used to connect to the database.

connectionLimit

The maximum number of concurrent connections to the database.

transactions

Boolean value. Should be set to true unless you use a database that is not capable of handling transactions like the MyISAM default table type of MySQL.

<typeMap>

Contains no attributes and seven child nodes. Each database may have its own mapping of JDBC datatypes to their internal datatypes. For MySQL the typeMap node looks as follows:

```
<tinyint      dbType="TINYINT"/>
<integer      dbType="INTEGER"/>
<bigint       dbType="BIGINT"/>
<double       dbType="DOUBLE"/>
<timestamp    dbType="DATETIME"/>
<varchar      dbType="VARCHAR(255)"/>
<longvarchar  dbType="TEXT"/>
```

The names of the nodes are the seven JDBC datatypes used by the Oregano Server. The dbType attribute specifies the corresponding MySQL datatype that will be used in `CREATE TABLE` statements. The datatype mappings will be verified when the server boots. If a mismatch is detected, an error description will be written to `bootLog.txt` and the booting will be aborted. If you manage to create a working type map for other databases than MySQL feel free to send it to `info@oregano-server.org`.

<statistics>

Contains one required attribute and no child nodes.

Example:

```
<statistics periodStart="00:00" />
```

periodStart

The time when the server should switch to the next date for all the statistical recordings it creates. The time must be specified in the format hh:mm. "00:00" is the most obvious value, but we tend to use "04:00" in our projects because at 4:00 most people are asleep. This way a login at 23:00 will be added to the same record as a login at 01:00, which we felt is the best way to detect nights with many visitors.

The statistical recordings that Oregano Server creates include the number of logins per day, the number of new registrations per day, the maximum number of concurrent connections per day and the average login time per user per day. It will be written to the statistics table in the database.

In addition the same time specified in this attribute will be used for the `executeDaily`, `executeWeekly` and `executeMonthly` methods of the server-side TaskManager class.

<services>

Contains six required attributes and no child nodes.

Example:

```
<services
     mailboxLimit="200"
     buddyLimit="100"
     blacklistLimit="100"
     groupListCache="30"
     groupListInterval="60"
     userListCache="30"
     userListInterval="60"
/>
```

mailboxLimit

The maximum number of mails in the in-box for each user.

buddyLimit

The maximum number of buddies each user can put into his buddylist.

blacklistLimit

The maximum number of users each user can put into his blacklist.

groupListCache

The number of seconds the list of groups will be cached. You can load the list with the `loadGroupList` method in the Info object of the client. If you enable caching, the server will cache the message containing the list of groups for subsequent requests until the specified number of seconds elapsed. To disable caching set the value to `0`.

groupListInterval

The length of the interval in seconds. The server sends the list of groups to all clients at regular intervals. It will be passed to the `onLoadGroupList` event handler of the Info object in the client. Set the value to `0` if you want to disable this feature and load the list of groups by hand with the `loadGroupList` method of the Info object.

userListCache

The number of seconds the list of users will be cached. You can load the list with the `loadUserList` method in the Info object of the client. If you enable caching, the server will cache the message containing the list of users for subsequent requests until the specified number of seconds elapsed. To disable caching set the value to `0`.

userListInterval

The length of the interval in seconds. The server sends the list of users to all clients at regular intervals. It will be passed to the `onLoadUserList` event handler of the Info object in the client. Set the value to `0` if you want to disable this feature and load the list of users by hand with the `loadUserList` method of the Info object.

<permissions>

Contains three required attributes and no child nodes.

Example:

```
<permissions
    changeEmail="true"
    changePassword="true"
    changePermissions="false"
/>
```

changeEmail

Boolean value. If set to `true` a client is permitted to change the email address with the `setEmail` method of the User object.

changePassword

Boolean value. If set to `true` a client is permitted to change the password with the `setPassword` method of the User object.

changePermissions

Boolean value. If set to `true` a client is permitted to change the permissions with the `setPermissions` method of the User object.

<groupConfig>

Contains one optional attribute, one loginGroup child node and an unlimited number of optional group nodes

Example:

```
<groupConfig defaultExtension="DefaultGroup">
    <loginGroup>
        <!-- see loginGroup section -->
    </loginGroup>
    <group>
        <!-- see group section -->
    </group>
    <group>
        <!-- see group section -->
    </group>
</groupConfig>
```

defaultExtension *

The name of the class to be used to extend the functionality of all groups. This attribute is optional. The specified class must implement `org.omus.ext.GroupExtension`. See the section on GroupExtension in the Server API chapter.

<loginGroup>

Contains one optional attribute and no child nodes. The login group is a special group that serves as an "entrance hall". Each user is member of this group after she logged in. The login group is different from normal groups in that you cannot close this group, cannot specify a user limit and cannot rejoin this group once you changed into another group. Another important difference is that you are "alone" as a member of the login group. You do not receive `onUserJoined` events in the client if another user logs in and you do not have access to the properties of other users in the login group. All these limitations were implemented to avoid bottlenecks in the login process. The actual interaction should take place in one of the other groups.

Example:

```
<loginGroup extension="LoginExt">
    <properties>
        <!-- see properties section -->
    </properties>
</loginGroup>
```

extension *

The name of the class to be used to extend the functionality of the login group. This attribute is optional. The specified class must implement `org.omus.ext.GroupExtension`. See the section on GroupExtension in the Server API chapter.

<group>

Contains two required attributes, one optional attribute, an unlimited number of optional `startup` child nodes and an optional `properties` child node.

Example:

```
<group
    configID="publicRoom"
    userLimit="25"
    extension="PublicExt"
    >
    <startup server="serv1" name="room_1" />
    <startup server="serv1" name="room_2" />
    <startup server="serv1" name="room_3" />
    <properties>
        <!-- see properties section -->
    </properties>
</group>
```

configID

An arbitrary string. The configID can be passed to the `change` method of the Group object in the client to specify which configuration to use to create the new group.

userLimit

The maximum number of users permitted to join the group.

extension *

The name of the class to be used to extend the functionality of all groups that were created with the id specified in the configID attribute. This attribute is optional. The specified class must implement `org.omus.ext.GroupExtension`. See the section on GroupExtension in the Server API chapter.

<startup>

Contains two required attributes and no child nodes. Each group node can contain an unlimited number of startup nodes. With each startup node you define a group that will be created when the server boots.

Example:

```
<startup server="serv1" name="room_1" />
```

server

The ID of the server that this group should be created on. Refers to an ID that you specified in the corresponding `server` node.

name

The name of the group. You can create an unlimited number of groups with the same configID but different names, but you cannot create two groups with the same name.

<properties>

Contains no attributes and an unlimited number of optional `groupProp` child nodes. Each group node can contain a `properties` node to define properties for groups created with a particular configID. In addition there is a top level `globalProperties` node in config.xml to define properties for all groups and for users.

Example:

```
<properties
    <groupProp>
        <!-- see groupProp section -->
    </groupProp>
    <groupProp>
        <!-- see groupProp section -->
    </groupProp>
</properties>
```

<globalProperties>

Contains no attributes and an unlimited number of optional `groupProp` or `userProp` child nodes. The group properties you define inside this node will be created for all groups. If you want to add group properties for a particular set of groups, define them in the properties node within the correspomding `group` node.

Example:

```
<globalProperties
    <userProp>
        <!-- see userProp section -->
    </userProp>
    <groupProp>
        <!-- see groupProp section -->
    </groupProp>
</globalProperties>
```


<groupProp>

Contains four required attributes, two optional attributes and no child nodes.

Example:

```
<groupProp
    name="itemList"
    type="largeTable"
    persistence="writeOnChange"
    cacheLevel="serverCache"
    definitionID="itemL"
/>
```

name

The name of the property

definitionID *

Applies to properties with a type attribute set to `table` or `largeTable` only. Refers to the `id` attribute of the corresponding `tableDefinition` node. For each table property you must define the name and datatype for each column in the `tableDefinition` node.

default *

The default value of this property. This attribute will be ignored for the container types (`array`, `object`, `table`) because they are always empty by default. If you specify the default value for a date property it must be the number of milliseconds from the epoch (midnight GMT, January 1st, 1970). If you are not in the mood for calculating the value, you can use the `getTime` method of a `java.util.Date` or the `getTime` method of an ActionScript `Date` object.

type

The datatype of the property. Must be one of the following values:

attribute value	ActionScript datatype	Java datatype
boolean	boolean	BooleanField
int	number	IntField
long	number	LongField
float	number	DoubleField
counter	number	CounterField
string [1)]	String	StringField
largeString [2)]	String	StringField
array [3)]	Array	DataList
largeArray [2)]	Array	DataList
object [3)]	Object	DataRow
largeObject [2)]	Object	DataRow
table [3)]	org.omus.Table	DataTable
largeTable [2)]	org.omus.Table	DataTable

[1)] max 255 characters

[2)] limit depends on the internal datatype of your database that maps to JDBCs longvarchar

[3)] max 255 characters for the marshalled value

For a description of the special Java datatypes used to map the corresponding ActionScript types see the section on package `org.omus.data` in the Server API chapter.

persistence

Must be one of the following values:

off	The value will not be stored in the database. Each time the group is created, the value of this property will be reset to the default value.
writeOnRemoval	The value will be stored in the database. Any modifications will be written to the database when the group is removed. (Dynamically created groups will be removed automatically when the last user leaves the group, groups created at startup will be removed when the server shuts down)
writeOnChange	The value will be stored in the database. Each modification will be written to the database immediately.

The fields required to store the properties will be created automatically in the tables `userprops` or `groupprops` respectively. On the other hand, if you remove a property from the configuration file, the corresponding column will not be removed in the database, thus, you do not risk losing data, if you boot Oregano Server with the wrong configuration file by mistake. The logging table will include an entry with a list of all columns that are obsolete, so that you can delete them manually.

cacheLevel

Must be one of the following values:

off	The value will not be cached in the clients nor in the server. It has to be read from the database each time it is accessed from client-side or server-side code with the load methods of the PropertySet object.
serverCache	The value will be loaded into the PropertySet object of the corresponding Group object in the server. This way you have quick access to the value if you write a server extension that reads or modifies the value frequently. The value will not be loaded into any client, thus the client-side PropertySet objects must load the value from the server-side object explicitly if they need to access it.
synchronizeGroup	The value will be loaded into the PropertySet object of the Group object in the server. Furthermore it will be loaded automatically into all the clients of the group. With this setting you have quick access to the value in server extensions and in all clients. Of course any changes will be reflected in all clients immediately.

<userProp>

Contains four required attributes, two optional attributes and no child nodes.

Example:

```
<userProp
    name="age"
    type="int"
    persistence="writeOnChange"
    cacheLevel="off"
    default="99"
/>
```

name

The name of the property

type

The datatype of the property. See the `type` attribute of the `groupProp` node for a list of valid values.

persistence

Must be one of the following values:

off	The value will not be stored in the database. Each time the user logs in, the value of this property will be reset to the default value.
writeOnLogout	The value will be stored in the database. Any modifications will be written to the database when the user logs out. Useful for values that change frequently.
writeOnChange	The value will be stored in the database. Each modification will be written to the database immediately.

The fields required to store the properties will be created automatically in the tables userprops or groupprops respectively. On the other hand, if you remove a property from the configuration file, the corresponding column will not be removed in the database, thus, you do not risk losing data, if you boot Oregano Server with the wrong configuration file by mistake. The logging table will include an entry with a list of all columns that are obsolete, so that you can delete them manually.

cacheLevel

Must be one of the following values:

off	The value will not be cached in the clients nor in the server. It has to be read from the database each time it is accessed from client-side or server-side code with the load methods of the PropertySet object.
serverCache	The value will be loaded into the PropertySet object of the corresponding User object in the server. This way you have quick access to the value if you write a server extension that reads or modifies the value frequently. The value will not be loaded into any client, thus the client-side PropertySet objects must load the value from the server-side object if they need to access it.
synchronizeClient	The value will be loaded into the PropertySet object of the corresponding User object in the server. Furthermore it will be loaded automatically into the client that the property is associated with. With this setting you have quick access to the value in server extensions and in the client that owns this property. Other clients in the group still have to explicitly load the value.
synchronizeGroup	The value will be loaded into the PropertySet object of the corresponding User object in the server. Furthermore it will be loaded automatically into all the clients of the group. With this setting you have quick access to the value in server extensions and in all clients.

definitionID *

Applies to properties with a type attribute set to `table` or `largeTable` only. Refers to the id attribute of the corresponding `tableDefinition` node. For each table property you must define the name and datatype for each column in the `tableDefinition` node.

default *

The default value of this property. This attribute will be ignored for the container types (`array`, `object`, `table`) because they are always empty by default. If you specify the default value for a date property it must be the number of milliseconds from the epoch (midnight GMT, January 1st, 1970). If you are not in the mood for calculating the value, you can use the `getTime` method of a `java.util.Date` or the `getTime` method of an ActionScript `Date` object.

<tableDefinition>

Contains one required attribute and one or more column child nodes. Needed if you configured one or more properties of type `table`. Each table definition can be used for more than one table property.

Example:

```
<tableDefinition id="playerList">
    <column name="name" type="string" />
    <column name="age" type="int" />
    <column name="score" type="int" />
    <column name="repository" type="array" />
</tableDefinition>
```

id

The ID of this table definition. Can be referred to in the `definitionID` attribute in a `userProp` or `groupProp` node.

<column>

Contains one required attribute and one or more column child nodes.

Example:

```
<column name="age" type="int"
```

name

The name of the column.

type

The datatype of the property. See the `type` attribute of the `groupProp` node for a list of valid values.

The only restriction is that you can not use the counter type in a table definition.

<log>

Contains one required attribute and 12 child nodes.

Example:

```
<log output="database">
    <login          level="warn" />
    <logout         level="warn" />
    <database       level="warn" />
    <messaging      level="error" />
    <groups         level="warn" />
    <changeGroups   level="warn" />
    <services       level="warn" />
    <admin          level="info" />
    <rmi            level="warn" />
    <client_omus    level="info" />
    <client_dev     level="warn" />
    <extensions     level="warn" />
</log>
```

output

One of the following two values:

database	all log entries are written to the logging table in the database
file	all log entries are written to log.txt. Most useful for debugging.

You can set a log-level for each of the 12 categories. Valid values for the level attribute in each node are `debug`, `info`, `warn` and `error`. The `debug` and `info` levels are useful for debugging but way to verbose if many users are logged in. In this case the most appropiate setting for each category is `warn`.

For 10 of these 12 categories log entries will be written by the server automatically. The only exceptions are `client_dev` and `extensions`. The former is the category that contains all log entries that you sent from a client using one of the methods of the Log object in the Client API. The latter contains all log entries that you created through calling one of the methods of the LogManager in the Server API. See the corresponding chapters for more details on logging.

dbCore.xml - SQL used by Oregano Server

This configuration file contains all SQL statements that are used by the Oregano Server. It has been tested with MySQL. If you use another database it may be possible that you have to adjust some of the SQL statements. Especially the `JOIN` syntax may differ in other database systems. Please do not modify any other nodes than the statement nodes. If you managed to adjust this configuration file for a different database feel free to send it to `info@oregano-server.org`.

dbCustom.xml - custom SQL statements

All the SQL statements that you want to use either with the ActionScript DbTransaction object of the Client API or the DbTransaction class of the Server API have to be configured in this file. There are two types of top-level nodes: `dbReader` and `dbWriter`. The former is intended for `SELECT` statements, the latter for `UPDATE`, `INSERT` or `DELETE` statements.

Attributes:

configID

The ID that will be used as a first parameter in every constructor for DbReader or DbWriter objects (client-side and server-side).

rollbackOnError

Boolean value. Set this attribute to `true` if you want the transaction to be rolled back when this statement caused an error. The transaction will simply be aborted, if your database does not support transactions.

minRows *

Integer value. This is an optional attribute. For dbWriter nodes it is the required minimum number of affected rows of the corresponding `DELETE`, `INSERT` or `UPDATE` statement, for dbReader nodes it is the required minimum number of rows in the result set. If the actual number of rows is lower than specified, the transaction will be rolled back (or aborted, if your database does not support transactions).

Child nodes:

statement

Contains the SQL statement. Oregano Server uses Prepared Statements so all the parameters are replaced by a ?.

in

The `in` nodes specify all the parameters that will be passed to the statement. Note that the order of the `in` nodes must correspond to the order of question marks in your statement

out

Can only be included in `dbReader` nodes. Useful if the specified value is required as an `in`-value in a subsequent statement.

The name attribute of an `out` node must correspond to one of the field names read by the statement.

See the section on the DbTransaction object in the Client API or Server API chapter for example code.

```
<dbWriter configID="updateCat" rollbackOnError="true" minRows="1">
    <in name="color" />
    <in name="catID" />
    <statement>
        UPDATE cats SET color = ? WHERE catID = ?
    </statement>
</dbWriter>

<dbReader configID="loadCat" rollbackOnError="true">
    <in name="catID" />
    <statement>
        SELECT * FROM cats WHERE catID = ?
    </statement>
</dbReader>
```

CHAPTER 5
Client API

Overview

The Client API is fully object-oriented. Each method and each event handler of each object is explained in the Reference section of this chapter.

To use the Client API you must include it in your Flash Movie. Copy the `oregano_as` directory of your Oregano Client download into the directory of your Flash file. Then add the line

```
#include "oregano_as/init.as"
```

to your movie. Right after the include statement it is advisable to initialize the Session object with a line like this:

```
org.omus.session.init("vers_2_12","www.myserver.com",1655);
```

For more details see the section on the Session object in the Reference part of this chapter; `org.omus` is the namespace for all Oregano objects.

To give you an overview of the Client API, here is a full list of all objects:

Session
Handles the connection to the server. Use this object to log in and log out a user.

Group
Represents the current group. Contains methods to close a group, to change into another group or to obtain the PropertySet objects of the group or a user currently in the group.

User
Represents the user that was logged in with this client. Other users of the current group are represented through their PropertySet objects.

Messenger, Message, MessageFilter
Use these object to easily exchange data between clients or between a client and the server.

PropertySet, PropertyLoader
These objects manage the properties associated with a group or a user. The PropertySet object handles all the persistence and synchronization requirements according to the configuration in config.xml.

Table, TableDefinition, Class

Table is an additional datatype that can be used as a property. Useful if you want to manage and synchronize large amounts of data, since it can be partially synchronized. Use the Class object to register your own custom ActionScript objects, so that they can be used in Properties or Messages.

Mailbox, Mail, Buddies, Info, Locks

Utility classes. Integrate email functionality into your application, easily manage the user's buddy list, or use the Info object to load a list of all users currently online or a list of all groups that were created. Or use the Locks object to lock objects in the current group in a synchronied way.

DbTransaction, DbWriter, DbReader, DbResult

Use these objects to read or write data to or from the database without the need to write server-side code.

Log, Error

A logging facility and an object that gets passed to each `onError` event handler.

Event Model

With Flash MX a new event model was introduced. If you are already familiar with that new model, you will quickly get used to the way the objects in the Oregano Client work. Most of those objects generate events, usually when they receive data or an error message from the server. To get notified of the event, a listener has to be registered with that object. The listener must contain methods for one or more of the events it is interested in. It can simply be a generic ActionScript object with an event handler method attached to it, or it can be one of your own custom classes.

The Messenger object for example generates an `onMessage` event each time it receives a message sent from the server. To handle incoming messages, you can write code like this.

```
var lis = new Object();
lis.onMessage = function (msg) {
    trace("received a message");
    trace("the subject is " + msg.getSubject());
};
org.omus.messenger.addListener(lis);
```

Now each time a message is received in the client, two lines will be added to the output window. You will find more examples of listeners in the Reference section of this chapter.

In addition to the regular events almost every object generates `onError` events passing an `org.omus.Error` object as an argument. You can use the methods of that error object to obtain the error code, an description of the error or the name of the method that caused the error.

Exchanging data

Building multiuser applications implies that data has to be exchanged between clients and the server. There are two approaches to accomplish this in Oregano Server: Using properties or using messages. To give you an impression on the difference of these two approaches, let's look at a small example:

In a visual chat application with an avatar representing each user, you want to visualize the movement of the avatars and exchange chat messages. The former is best done using a property, the latter is a typical job of a Message object.

Message objects are transient. They can send data from one client to all the other clients, to one particular client or to the server. The other clients receive those messages in their `onMessage` event handler of the messenger object. After the message was received, the client and the server forget about it. This suits well for the task of sending chat messages because they are only needed at the moment they are received. If a new user enters the room later, she might not be interested in reading all the stuff that was said hours ago.

The position of an avatar, on the other hand, must be sent to each other client including those who join later. This can be accomplished with a property. The server keeps track of changes and if a new user enters a group, the server sends the current values for all properties of the group and all properties of the other users in the group to the client. (Well, there are exceptions, depending on the configuration of the properties, but let's keep this example simple) Properties handle all the persistence and synchronization stuff for you. The only extra work that you have to do is to define each property you want to use in config.xml.

Let's add some code. To send the chat message to other clients, you write code like this:

```
sendChatMsg = function (chatMsg) {
    var msg = new org.omus.Message("chat message");
    var attach = msg.getAttachment();
    attach.chatText = chatMsg;
    org.omus.messenger.sendToGroup(msg);
};
```

The other clients receive this message in the `onMessage` event handler of the Messenger object. You must add a listener to handle the message:

```
var lis = new Object();
lis.onMessage = function (msg) {
    if (msg.getSubject() == "chat message") {
        var chatMsg = msg.getAttachment().chatText;
        var sender = msg.getSender();
        // display chatMsg
    }
};
org.omus.messenger.addListener(lis);
```

When an avatar changed its position you must modify the value of the corresponding property:

```
changePosition = function (username, xpos, ypos) {
    var propSet = org.omus.group.getUserProperties(username);
    propSet.setValue("position",{x:xpos,y:ypos});
    propSet.synchronize();
};
```

For the other clients to be notified they must have a listener for `onSynchronize` events registered with the PropertySet object:

```
var lis = new Object();
lis.onSynchronize = function (newProps, clientRequest) {
    if (newProps.contains("position")) {
        var pos = newProps.getValue("position");
        var newXpos = pos.x
        var newYpos = pos.y
        // move avatar to new position
    }
};
// add this listener to the PropertySet object of each user!!
```

Before you can use the position property, you must define it in config.xml:

```
<globalProperties>
    <userProp
        name="position"
        type="object"
        persistence="off"
        cacheLevel="synchronizeGroup"
    />
    <!-- other properties .... -->
</globalProperties>
```

The persistence setting is "off" because if the user logs out nobody is interested in remembering her last position. The cacheLevel is set to "synchronizeGroup" so that the value is synchronized in all clients of the group automatically.

The org.omus namespace

The namespace `org.omus` is used for all the objects that are created automatically. It is attached to the `_global` object in Flash 6 players and to `MovieScript.prototype` in Flash 5 players. This way it can be referenced the same way in both players. The Session object for example is one of these singletons and thus can be referenced with `org.omus.session`. Three objects will be created immediately when you include init.as:

org.omus.session
org.omus.clazz
org.omus.log

The session object is used to connect to the server and the class object is used to register your own custom ActionScript classes, so that they can be exchanged between

clients. After the user has logged in, four more objects are created in the `org.omus` namespace:

org.omus.group
org.omus.user
org.omus.messenger
org.omus.info

All the other objects must be either created by hand or must be obtained through a method of another object. If you create a client side Oregano object, that has a public constructor, you must also use the namespace for the constructor as follows:

```
var pl = new org.omus.PropertyLoader();
```

Reference

The reference contains only the public methods of each object. If you look into the ActionScript include files, you will find many methods and classes that are not mentioned in the reference. Using one of those methods in your application might lead to unexpected results. Unfortunately ActionScript methods do not have access modifiers like Java methods, so it was impossible to declare them as private. But as long as you use only the documented methods you are on the safe side.

Each section in the reference describes one object. It first gives you an overview of all the methods and event handlers available in that object and then describes each method in detail. The notation of each method entry looks as follows:

getUserProperties

```
PropertySet getUserProperties (String username)
```

This may seem a bit unfamiliar for ActionScript programmers. It includes the datatype of the return value and the datatypes of all arguments. Of course ActionScript is a loosely typed language, but the Oregano Client API is not! When you pass an invalid datatype to a method of one of the objects in the Oregano Client, an error will be generated and sent to the server as a log entry. There are very few exceptions, for example the `getValue` method of the PropertySet object which can return any kind of datatype. In these cases the return type is `any`. If the method does not return anything the return type is `void`. For arguments and return values of type `Array`, the datatype for the elements of that array is indicated as follows: `Array[String]`. This example specifies an array that can only contain elements of type `String`.

Buddies

This object manages a list of buddies. Each user can have his own buddy list which will be stored in the database. You obtain a reference to the Buddies object with `org.omus.user.getBuddies()`.

Method Summary

void **addListener** (Object listener)
void **insert** (String username)
void **load** ()
void **remove** (String username)
void **removeAll** ()
void **removeListener** (Object listener)

Event Summary

onError (Error error)
onInsert (String username)
onLoad (Array[String] usernames)
onRemove (String username)
onRemoveAll ()

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the Buddies object can generate.

insert

```
void insert (String username)
```

Inserts a new user into the buddy list.

load

```
void load ()
```

Loads the buddy list into the client.

remove

```
void remove (String username)
```

Removes a buddy from the list

removeAll

```
void removeAll ()
```

Removes all buddies from the list.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured that was caused by one of the methods of the Buddies object.

onInsert

```
onInsert (String username)
```

Event handler; invoked when a buddy was inserted.

onLoad

```
onLoad (Array[String] usernames)
```

Event handler; invoked when the buddy list was loaded.

onRemove

```
onRemove (String username)
```

Event handler; invoked when a buddy was removed.

onRemoveAll

```
onRemoveAll ()
```

Event handler; invoked when all buddies were removed.

Class

The class object contains methods to register your custom ActionScript classes so that objects of that class can be exchanged between clients. After you registered one of your custom classes, you can use objects of that class as a value of a user or group property or as part of an attachment of a Mail or Message object. It ensures that your objects will be unmarshalled in other clients with all their properties and methods. Please note that only methods that you added to the prototype object of the constructor can be attached to the unmarshalled object.

The Class object is a singleton that will be created automatically. You can reference it with `org.omus.clazz`.

Method Summary

Function **getConstructor** (String name)
void **register** (String name, Function constructor)

getConstructor

```
Function getConstructor (String name)
```

Returns the constructor that was registered with the specified name.

register

```
void register (String name, Function constructor)
```

Registers the specified constructor with an arbitrary name.

DbReader

This object represents a `SELECT` statement to be executed on the server.

Constructor Summary

new **DbReader** (String configID *[, String resultID]*)

Method Summary

void **setParam** (String name, any value)

DbReader

```
new org.omus.DbReader (String configID [, String resultID])
```

Creates a new DbReader object. The specified configID must match a configID attribute in a `<dbReader>` node in dbCustom.xml. The optional resultID must be specified if you want to read values from the DbResult object. The same resultID must then be passed to the `getField` or `getTable` methods of the DbResult object. The resultID is necessary because you can include an unlimited number of DbReader objects in a single DbtTransaction object.

setParam

```
void setParam (String name, any value)
```

Sets the parameter with the specified name to the specified value. The name of the parameter must match the name attribute of one of the `<in>` nodes you specified in the `dbCustom.xml` file. The value can be one of the following datatypes: `boolean`, `number`, `string`, `Date`, `Array`, `Object`, `org.omus.Table`.

DbResult

This object represents the result of a transaction. It will be passed to the `onResult` event handler of the DbTransaction object.

Method Summary

int **getAffectedRows** (String resultID)
Table **getTable** (String resultID)

getAffectedRows

```
int getAffectedRows (String resultID)
```

Returns the number of affected rows of the statement with the specified resultID. The resultID must match the ID that you passed as the second argument to the constructor of the corresponding DbWriter object.

getTable

```
org.omus.Table getTable (String resultID)
```

Returns a reference to a table object. The resultID must match the ID that you passed as the second argument to the constructor of the corresponding DbReader object.

DbTransaction

This object can be used to read and write to and from the database. Note that persistent properties will be written to the database automatically. You only need this object if you created additional tables in the database.

This object is useful if you want to execute custom SQL statements. For security reasons you cannot construct those statements in the client. Instead you must configure them in the dbCustom.xml file on the server. This approach may seem a bit cumbersome at first sight. But it is very versatile once you get used to it.

The following example illustrates how you can use the DbTransaction object to read a user ID from one table and use that ID to change a value in another table without contacting the server twice.

The corresponding entries in dbCustom.xml might look as follows (see the Configuration chapter for more details):

```
<dbReader configID="readUserID" rollbackOnError="true" minRows="1">
    <in name="username" />
    <out name="userID" />
    <statement>
        SELECT userID FROM reginfo WHERE username = ?
    </statement>
</dbReader>

<dbWriter configID="updateAddress" rollbackOnError="true">
    <in name="street" />
    <in name="city" />
    <in name="phone" />
    <in name="userID" />
    <statement>
        UPDATE address SET street = ?, city = ?, phone = ? WHERE userID = ?
    </statement>
</dbWriter>
```

The ActionScript code to execute those statements looks as follows:

```
var ta = new org.omus.DbTransaction();
ta.addPart(new org.omus.DbReader("readUserID"));
ta.addPart(new org.omus.DbWriter("updateAddress","address"));
ta.setParam("username","Thomas");
ta.setParam("street","217 Mulholland Drive");
ta.setParam("city","Los Angeles");
ta.setParam("phone","27 27 55 55");

var lis = new Object();
lis.onResult = function (result) {
    trace("affected rows = " + result.getAffectedRows("address"));
};
lis.onError = function (error) {
    trace ("ERROR: " + error.getDescription());
};
ta.addListener(lis);
ta.execute();
```

Please note that the second statement has 4 `<in>` nodes. Three of them (street, city and phone) will be set by your ActionScript code. The last one (userID) will be set automatically because it is configured as an `<out>`-value in the first statement which will be available to all subsequent statements. The attribute minRows is set to "1" in the first statement so that execution of the transaction will be aborted if the result is empty, because the second statement would fail anyway, if no user with the specified name exists.

Constructor Summary

new **DbTransaction** ()

Method Summary

void **addListener** (Object listener)
void **addPart** (DbReader|DbWriter part)
void **execute** ()
void **removeListener** (Object listener)
void **setParam** (String name, any value)

Event Summary

onResult (DbResult result)
onError (Error error)

DbTransaction

```
new org.omus.DbTransaction ()
```

Creates a new DbTransaction object.

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the DbTransaction object can generate.

setParam

```
void setParam (String name, any value)
```

Sets the parameter with the specified name to the specified value. The name of the parameter must match the name attribute of one of the <in> nodes you specified in the dbCustom file. The value can be one of the following datatypes: `boolean`, `number`, `string`, `Date`, `Array`, `Object`, `org.omus.Table`. If you set a parameter in the DbTransaction object it is available to all the DbWriter or DbReader parts you add to this transaction. If you want to set parameters that can only be read by one particular DbWriter or DbReader object, you must use the `setParam` method of those objects.

addPart

```
void addPart (DbReader|DbWriter part)
```

Adds a DbReader or DbWriter object to this transaction.

execute

```
void execute ()
```

Executes this transaction. The result will be passed to the `onResult` event.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured that was caused by the `execute` method of this object.

onResult

```
onResult (DbResult result)
```

Event handler; invoked when the result of the transaction has been received.

DbWriter

This object represents a `DELETE`, `INSERT` or `UPDATE` statement to be executed on the server.

Constructor Summary

new **DbWriter** (String configID *[, String resultID]*)

Method Summary

void **setParam** (String name, any value)

DbWriter

```
new org.omus.DbWriter (String configID [, String resultID])
```

Creates a new DbWriter object. The specified configID must match a configID attribute in a <dbWriter> node in dbCustom.xml. The optional resultID must be specified if you want to check the number of affected rows in the DbResult object.

setParam

```
void setParam (String name, any value)
```

Sets the parameter with the specified name to the specified value. The name of the parameter must match the name attribute of one of the <in> nodes you specified in the dbCustom file. The value can be one of the following datatypes: `boolean`, `number`, `string`, `Date`, `Array`, `Object`, `org.omus.Table`.

Error

An error object gets passed to all the `onError` event handlers. You can use it to retrieve detailed information about the error.

Method Summary

Array **getArguments** ()
String **getCode** ()
String **getDescription** ()
String **getMethodName** ()

getArguments

```
Array getArguments ()
```

Returns an array of all the arguments that were passed to the method that caused the error. May return `null` if the error was caused by the server.

getCode

```
String getCode ()
```

Returns the error code of that object.

getDescription

```
String getDescription ()
```

Returns a description of the error that occured.

getMethodName

```
String getMethodName ()
```

Returns the name of the method that caused the error. May return `null` if the error was caused by the server.

Group

The group object represents the group that the client is currently member of. Each client can only be member of one group at any point in time. The methods of the group object can be used to obtain information about the current group and to retrieve the PropertySet object of the group or of any user currently in this group. Note that all groups must be configured in the groupConfig node in config.xml on the server. See the Configuration chapter for details.

The Group object is a singleton that will be created automatically when a user logs in. It can be referenced with `org.omus.group`.

Method Summary

void **addListener** (Object listener)
void **change** (String groupName, String configID, boolean syncProps)
void **close** ()
boolean **containsUser** (String username)
Array[PropertySet] **getAllUserProperties** ()
String **getConfigID** ()
String **getName** ()
PropertySet **getProperties** ()
int **getUserCount** ()
int **getUserLimit** ()
PropertySet **getUserProperties** (String username)
boolean **isClosed** ()
boolean **isFull** ()
void **open** ()
void **removeListener** (Object listener)

Event Summary

onChange (Object data)
onClose ()
onError (Error error)
onOpen ()
onUserJoined (PropertySet userProps)
onUserLeft (PropertySet userProps)

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the Group object can generate.

change

```
void change (String groupName, String configID, boolean syncProps)
```

Attempts to leave the current group and change into another group; configID must correspond to an id that you used in a `<group>` node in config.xml on the server. You can pass `null` as the configID, but then the change only succeeds if a group with the specified name already exists. If you pass a valid configID the group will be created automatically on the server if it does not exist yet. Note that you can dynamically create an unlimited number of groups with the same configID. We get a lot of feedback from users who think that you cannot create groups dynamically because you have to configure them on the server. But each configID corresponds to **one** configuration that will be shared by an unlimited number of groups with arbitrary names.

The third parameter can be useful if you want to set some initial values for properties of this user which would make only sense if the change into the new group succeeds. Just change the values of the corresponding properties but do not call `PropertySet.synchronize`. Instead pass `true` as the third parameter to this method. If the change succeeds the new values will be accepted and the `PropertySet.onSynchronize` event handler will be invoked in addition to the `Group.onChange` event handler. If the attempt to change groups fails, all affected properties will be reset.

close

```
void close ()
```

Closes the group so that no other users can join.

containsUser

```
boolean containsUser (String username)
```

Returns `true` if the user with the specified name is a member of this group.

getAllUserProperties

```
Array[PropertySet] getAllUserProperties ()
```

Returns an array containing the PropertySet objects for each user that is currently member of the group. The array is sorted alphabetically by user names.

getConfigID

```
String getConfigID ()
```

Returns the configID of the group. It identifies the corresponding `<group>`-node in config.xml that was used to configure this group. You can create an unlimited number of groups with the same configID but different names. See the Configuration chapter for more details.

getName

```
String getName ()
```

Returns the name of the group.

getProperties

```
PropertySet getProperties ()
```

Returns the PropertySet object associated with this group.

getUserCount

```
int getUserCount ()
```

Returns the number of users who are currently member of this group.

getUserLimit

```
int getUserLimit ()
```

Returns the maximum number of users permitted for this group.

getUserProperties

```
PropertySet getUserProperties (String username)
```

Returns the PropertySet object associated with a user who is currently member of this group. Returns `null` if no user with the specified name exists in this group.

isClosed

```
boolean isClosed ()
```

Returns `true` if the group is closed so that no other users can join.

isFull

```
boolean isFull ()
```

Returns `true` if the maximum number of users has been reached.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

open

```
void open ()
```

Opens the group for other users to join.

onClose

```
onClose ()
```

Event handler; invoked when the group was closed so that no other user can join.

onChange

```
onChange (Object data)
```

Event handler; invoked when the attempt to change groups succeeded. When this event handler is invoked the group object already represents the new group. The argument is an empty object unless you wrote a GroupExtension as a server side plugin which can be used to send additional data to the client.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured that was caused by one of the following methods of the Group object: `change`, `open`, `close`.

onUserJoined

```
onUserJoined (PropertySet userProps)
```

Event handler; invoked when a user joined this group. The argument is the PropertySet object that represents the new user.

onUserLeft

```
onUserLeft (PropertySet userProps)
```

Event handler; invoked when a user left this group. The argument is the PropertySet object that represents the user that has left the group.

onOpen

```
onOpen ()
```

Event handler; invoked when the group was opened so that other users can join.

Info

Basically a small collection of methods to retrieve information that did not fit into any other object.

The Info object is a singleton that will be created automatically when a user logs in. It can be referenced with `org.omus.info`.

Method Summary

void **addListener** (Object listener)
void **findUser** (String username)
void **loadGroupList** ()
void **loadUserList** ()
void **removeListener** (Object listener)

Event Summary

onError (Error error)
onFindUser (String username, String groupName)
onLoadGroupList (Table groups)
onLoadUserList (Array[String] usernames)

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the Info object can generate.

findUser

```
void findUser (String username)
```

Locates the user with the specified name. The `onFindUser` event handler will return the name of the group, that the user with the specified name is currently member of or `null` if the user is offline.

loadGroupList

```
void loadGroupList ()
```

Loads a table containing all the groups that were created on the server. If you need the list of groups in regular intervals you can set `groupListInterval` in the `<services>` node in the server configuration to the number of seconds between updates. In this case you do not need to call this method, the `onLoadGroupList` event handler will be called automatically.

loadUserList

```
void loadUserList ()
```

Loads an array of the names of all users that are currently logged in. If you need the list of users in regular intervals you can set `userListInterval` in the `<services>` node in the server configuration to the number of seconds between updates. In this case you do not need to call this method, the `onLoadUserList` event handler will be called automatically.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured that was caused by one of the methods of the info object.

onFindUser

```
onFindUser (String username, String groupName)
```

Event handler; invoked when a user was located; groupName may be `null` if the user is currently offline or does not exist.

onLoadGroupList

```
onLoadGroupList (Table groups)
```

Event handler; invoked when the group list was loaded either due to a call to `loadGroupList` or due to a regular update sent by the server. Each row in the table contains information about a single group currently active on the server. The columns are: `name (String)`, `userCount (number)`, `userLimit (number)` and `closed (boolean)`.

onLoadUserList

```
onLoadUserList (Array[String] usernames)
```

Event handler; invoked when the user list was loaded either due to a call to `loadUserList` or due to a regular update sent by the server.

Locks

The Locks object can be used to restrict access to certain items within a group. The locks are valid within one group only, if a user leaves a group, all locks that he previously aqcuired will be released automatically. You obtain a reference to the Locks object with `org.omus.user.getLocks()`. The names of the locks are arbitrary strings.

An example is a whiteboard application with some draggable objects on the screen. To prevent that two users start a drag operation concurrently, you can use a lock for each draggable item on the screen. Example code to set up a listener for the locks object might look as follows:

```
var locks = org.omus.user.getLocks();
var lis = new Object();
```

```
lis.onAcquire = function (lockName, success) {
     if (success) {
          if (lockName == "draggableCircle") {
               // start drag operation for circle
          } else if (lockName == "draggableSquare") {
               // start drag operation for square
          }
     } else {
          // show error message or play sound...
     }
};
locks.addListener(lis);
```

Method Summary

void **acquire** (String lockName)
void **addListener** (Object listener)
Array[String] **getAllAcquired** ()
boolean **isAcquired** (String lockName)
void **release** (String lockName)
void **releaseAll** ()
void **removeListener** (Object listener)

Event Summary

onAcquire (String lockName, boolean success)
onError (Error error)
onRelease (String lockName)
onReleaseAll ()

acquire

```
void acquire (String lockName)
```

Tries to acquire the lock with the specified name.

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the Locks object can generate.

getAllAcquired

```
Array[String] getAllAcquired ()
```

Returns a list of all locks currently held by this client.

isAcquired

```
boolean isAcquired (String lockName)
```

Returns true if a lock with the specified name is currently held by this client.

release

```
void release (String lockName)
```

Releases the lock with the specified name.

releaseAll

```
void releaseAll ()
```

Releases all locks.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

onAcquire

```
onAcquire (String lockName, boolean success)
```

Event handler; invoked when the acquisition of a lock was accepted or rejected, depending on the value of the second argument. The lock can only be acquired if no other user currently holds the same lock.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured that was caused by one of the methods of the Locks object.

onRelease

```
onRelease (String lockName)
```

Event handler; invoked when a lock was released.

onReleaseAll

```
onReleaseAll ()
```

Event handler; invoked when all locks were released.

Log

The log object contains methods to create custom log entries in the logging table on the server. The object uses four different log-levels: `DEBUG`, `INFO`, `WARN` and `ERROR`. A log-level is an indication of the importance of a message. Depending on the mininum log-level you specified for the `<oregano-dev>`-node in the logging section in config.xml a log entry might be ignored or sent to server.

If the configuration looks like this

```
<client_dev  level="warn" />
```

all log entries of level `WARN` or higher will be sent to the server. Accordingly calling `org.omus.log.debug()` or `org.omus.log.info()` will be ignored in this case.

The Log object is a singleton that will be created automatically when a user logs in. It can be referenced with `org.omus.log`.

Method Summary

void **addListener** (Object listener)
void **debug** (String errorCode, String info)
boolean **debugEnabled** ()
void **error** (String errorCode, String info)
boolean **errorEnabled** ()
void **info** (String errorCode, String info)
boolean **infoEnabled** ()
void **removeListener** (Object listener)
void **warn** (String errorCode, String info)
boolean **warnEnabled** ()

Event Summary

onLog (String log)

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for the `onLog` event.

debug

```
void debug (String errorCode, String info)
```

Creates a log entry with the specified error code and info string if the log object is configured to process messages of level `DEBUG` or higher. The error code will be automatically prefixed with `"cld-"` to avoid confusion with built-in error codes.

debugEnabled

```
boolean debugEnabled ()
```

Returns true if the log object is configured to sent log entries of level `DEBUG` or higher to the server.

error

```
void error (String errorCode, String info)
```

Creates a log entry with the specified error code and info string if the log object is configured to process messages of level `ERROR`. The error code will be automatically prefixed with `"cld-"` to avoid confusion with built-in error codes.

errorEnabled

```
boolean errorEnabled ()
```

Returns true if the log object is configured to sent log entries of level `ERROR` to the server.

info

```
void info (String errorCode, String info)
```

Creates a log entry with the specified error code and info string if the log object is configured to process messages of level `INFO` or higher. The error code will be automatically prefixed with `"cld-"` to avoid confusion with built-in error codes.

infoEnabled

```
boolean infoEnabled ()
```

Returns true if the log object is configured to sent log entries of level `INFO` or higher to the server.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

warn

```
void warn (String errorCode, String info)
```

Creates a log entry with the specified error code and info string if the log object is configured to process messages of level `WARN` or higher. The error code will be automatically prefixed with `"cld-"` to avoid confusion with built-in error codes.

warnEnabled

```
boolean warnEnabled ()
```

Returns true if the log object is configured to sent log entries of level `WARN` or higher to the server.

onLog

```
onLog (String log)
```

Event handler; invoked when logging information will be displayed in the output window. The parameter contains the whole formatted string that will be sent to the output window in your Flash authoring environment. If you test your Flash movies inside a browser you can use this event handler to send the logging information to a popup window or display it in your Flash movie.

Mail

Each Mail object represents a single mail. It has a subject, a sender and a recipient like normal internet email has too. But it does not have a body so all the text that you want to transmit with your mail has to be put into the attachment. But the attachment is not restricted to string objects. It is a normal ActionScript object and can contain any of the following datatypes: `boolean`, `number`, `string`, `Date`, `Object`, `Array`, `org.omus.Table`. You can write code to construct and send a mail as follows:

```
var mail = new org.omus.Mail("personal information");
var attach = mail.getAttachment();
attach.name = "Brian";
attach.age = 37;
attach.married = false;
attach.children = ["Jenny","Maria","Ben"];
var listener = new Object();
listener.onSend = function (mail) {
    trace("mail has been sent");
};
listener.onError = function (mail, error) {
    trace("sending mail failed: " + error.getDescription());
};
mail.addListener(listener);
mail.send("Jennifer");
```

Constructor Summary

new org.omus.**Mail** (String subject)

Method Summary

void **addListener** (Object listener)
Object **getAttachment** ()

Date **getDate** ()
Mailbox **getMailbox** ()
String **getRecipient** ()
String **getSender** ()
String **getSubject** ()
boolean **isLoaded** ()
boolean **isUnread** ()
void **loadAttachment** ()
void **mark** (boolean asRead)
void **remove** ()
void **removeListener** (Object listener)
void **send** (String recipient)

Event Summary

onError (Mail mail, Error error)
onLoadAttachment (Mail mail)
onMark (Mail mail)
onRemove (Mail mail)
onSend (Mail mail)

new

```
new org.omus.Mail (String subject)
```

Constructs a new Mail object with the specified subject.

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the Mail object can generate.

getAttachment

```
Object getAttachment ()
```

Returns the attachment of this mail object or `null` if the attachment has not been loaded into the client yet. It is a plain ActionScript object that is empty in a newly created Mail object. You can add properties with one of the following datatypes: `boolean`, `number`, `String`, `Date`, `Array`, `Object`, `org.omus.Table`.

getDate

```
Date getDate ()
```

Returns an ActionScript Date object representing the time this Mail object was sent.

getMailbox

```
org.omus.Mailbox getMailbox ()
```

Returns one of the two Mailbox objects that this mail belongs to.

getRecipient

```
String getRecipient ()
```

Returns the recipient of this Mail object. May return `null` if it is a new Mail object that has not been sent yet.

getSender

```
String getSender ()
```

Returns the sender of this Mail object.

getSubject

```
String getSubject ()
```

Returns the subject of this Mail object.

isLoaded

```
boolean isLoaded ()
```

Returns `true` if the attachment of this mail has been loaded.

isUnread

```
boolean isUnread ()
```

Returns `true` if this mail is unread.

loadAttachment

```
void loadAttachment ()
```

Loads the attachment for this Mail object.

mark

```
void mark (boolean asRead)
```

Marks this Mail object as read or as unread.

remove

```
void remove ()
```

Removes this Mail object from its containing Mailbox object.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

send

```
void send (String recipient)
```

Sends this Mail object to the specified recipient. You can send the same Mail object to more than one user. Each time you send a mail, a copy of this Mail object will be placed into the out box object.

onError

```
onError (Mail mail, Error error)
```

Event handler; invoked when an error occured that was caused by one of the following methods of the Mail object: `send`, `loadAttachment`, `mark`, `remove`.

onLoadAttachment

```
onLoadAttachment (Mail mail)
```

Event handler; invoked when the attachment has been loaded into this Mail object.

onMark

```
onMark (Mail mail)
```

Event handler; invoked when this Mail object has been marked as read or as unread.

onRemove

```
onRemove (Mail mail)
```

Event handler; invoked when this Mail object has been removed from its containing Mailbox object.

onSend

```
onSend (Mail mail)
```

Event handler; invoked when a Mail object has been sent successfully. Note that the argument is a copy of the original Mail object (the one you invoked the `send` method

on). This copy was put into the out box, because the original Mail object could have been sent to multiple recipients.

Mailbox

The Mailbox object stores Mail objects which are different from Message objects in that they are persistent. If you send a Mail object to a user who is currently offline, the Mail object will be delivered automatically the next time the recipient logs in. Each client has two Mailbox objects, one for the out box and one for the in box. Use the methods of the User object to obtain a reference to a Mailbox object.

This object is intended to implement some kind of email functionality in your multiuser application. If you need to send messages to exchange data between clients or between a client and the server as a requirement of your application logic, use Message objects which are similar to Mail objects. They differ in that they are not persistent and that they can be sent not only to a user, but also to a group, to all users currently online or to a server extension.

Method Summary

void **addListener** (Object listener)
void **load** ()
int **getLoaded** ()
Mail **getMail** (int index)
int **getTotal** ()
String **getType** ()
int **getUnread** ()
void **removeAll** ()
void **removeListener** (Object listener)
void **sortByDate** (boolean ascending)
void **sortBySender** (boolean ascending)
void **sortBySubject** (boolean ascending)

Event Summary

onError (Error error)
onLoad (Mailbox box)
onNewMail (Mailbox box)
onRemoveAll (Mailbox box)

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the Mailbox object can generate.

load

```
void load ()
```

Loads all Mail objects (without their attachments) that have not been loaded yet.

getLoaded

```
int getLoaded ()
```

Returns the number of loaded Mail objects in this mailbox.

getMail

```
Mail getMail (int index)
```

Returns the Mail object at the specified index. The mails are sorted by date per default; in this case the latest mail will be at index position 0. But you can change the order with one of the sort methods.

getTotal

```
int getTotal ()
```

Returns the total number of Mail objects in this mailbox.

getType

```
String getType ()
```

Returns either "inBox" or "outBox"

getUnread

```
int getUnread ()
```

Returns the number of unread Mail objects in this mailbox.

removeAll

```
void removeAll ()
```

Removes all Mail objects from this mailbox.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

sortByDate

```
void sortByDate (boolean ascending)
```

Sorts all Mail objects by date.

sortBySender

```
void sortBySender (boolean ascending)
```

Sorts all Mail objects by sender.

sortBySubject

```
void sortBySubject (boolean ascending)
```

Sorts all Mail objects by subject.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured that was caused by one of the following methods of the Mailbox object: `load`, `removeAll`.

onLoad

```
onLoad (Mailbox box)
```

Event handler; invoked when all Mail objects have been loaded into the Mailbox object.

onNewMail

```
onNewMail (Mailbox box)
```

Event handler; invoked when a new mail was received by the server. The corresponding Mail object will not be loaded automatically, thus `getLoaded` and `getTotal` will return different numbers after this event handler was invoked. You can use the `load` method at any time to load all new Mail objects into the client.

onRemoveAll

```
onRemoveAll (Mailbox box)
```

Event handler; invoked when all Mail objects have been removed from the Mailbox object.

Message

The Message object has only a small subset of the methods of the Mail object. Message objects are different from Mail objects in that they are not persistent. They can be sent using one of the methods of `org.omus.messenger`. The attachment is a normal ActionScript object and can carry any of the following datatypes: `boolean`, `number`, `String`, `Date`, `Object`, `Array`, `org.omus.Table`. You can write code to construct and send a message as follows:

```
var msg = new org.omus.Message("chat message");
var attach = msg.getAttachment();
attach.chatText = _root.getInputFieldContentSomehow();
org.omus.messenger.sendToGroup(msg);
```

Cunstructor Summary

new org.omus.**Message** (String subject)

Method Summary

Object **getAttachment** ()
String **getSender** ()
String **getSubject** ()

new

`new org.omus.Message (String subject)`

Constructs a new Message object with the specified subject.

getAttachment

`Object getAttachment ()`

Returns the attachment of this Message object. It is a plain ActionScript object that is empty in a newly created Message object. You can add properties with one of the following datatypes: `boolean`, `number`, `String`, `Date`, `Array`, `Object`, `org.omus.Table`.

getSender

`String getSender ()`

Returns the sender of this Message object.

getSubject

`String getSubject ()`

Returns the subject of this Message object.

MessageFilter

If you want fine grained control of which objects handle which kind of incoming messages, you can register a MessageFilter object with the messenger object. You can add an unlimited number of subjects and an unlimited number of listeners to each MessageFilter. When a message is received that has a subject that was added to a MessageFilter object, all the registered listeners in that MessageFilter receive an `onMessage` event.

In this example a filter is created that handles all messages with the subject `"start game"` or `"stop game"`:

```
var obj = new Object();
obj.onMessage = function (msg) {
    var at = msg.getAttachment();
    trace(at.newScore);
};
var filter = new org.omus.MessageFilter();
filter.addSubject("start game");
filter.addSubject("stop game");
filter.addListener(obj);
org.omus.messenger.addFilter(filter);
```

If you combine MessageFilter objects with the publish/subscribe mechanism in the Messenger object, you get precise control over the way messages are distributed and processed.

Constructor Summary

new org.omus.**MessageFilter** ()

Method Summary

void **addListener** (Object listener)
void **addSubject** (String subject)
void **removeListener** (Object listener)
void **removeSubject** (String subject)

new

`new org.omus.MessageFilter ()`

Constructs a new MessageFilter object with the specified subject.

addListener

`void addListener (Object listener)`

Adds the specified object as a listener to this MessageFilter. The listener object will receive an `onMessage` event for all incoming messages, that have a subject that was added to this filter with the `addSubject` method.

addSubject

```
void addSubject (String subject)
```

Adds the specified subject to this MessageFilter.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

removeSubject

```
void removeSubject (String subject)
```

Removes the specified subject from this MessageFilter.

Messenger

The Messenger object handles receiving, sending and filtering of Message objects which are different from Mail objects in that they are not persistent. If the recipient is offline the message will be discarded. There is a publish/subscribe mechanism which you can use to easily control the distribution of messages.

The Messenger object is a singleton that will be created automatically when a user logs in. It can be referenced with `org.omus.messenger`.

Method Summary

- void **addListener** (Object listener)
- void **addFilter** (MessageFilter filter)
- Array[String] **getSubscriptions** ()
- boolean **isSubscribed** (String subject)
- void **publish** (Message msg)
- void **removeListener** (Object listener)
- void **removeFilter** (MessageFilter filter)
- void **sendToAll** (Message msg)
- void **sendToGroup** (Message msg *[, String groupName]*)
- void **sendToServer** (Message msg)
- void **sendToUser** (Message msg, String username, boolean sameGroup)
- void **subscribe** (String subject)
- void **unsubscribe** (String subject)
- void **unsubscribeAll** ()

Event Summary

- **onError** (Error error)
- **onMessage** (Message msg)
- **onSubscribe** (String subject)
- **onUnsubscribe** (String subject)
- **onUnsubscribeAll** ()

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all events the Messenger object can generate.

addFilter

```
void addFilter (MessageFilter filter)
```

Adds a filter to the messenger object. Filtering may be convenient if you want fine grained control of which objects handle which kind of messages. Without filtering all incoming messages will be passed to the `onMessage` event handler of the Messenger object. In a large application this approach may be inappropiate. See the section on the MessageFilter object for details. You can register an unlimited number of filters.

getSubscriptions

```
Array[String] getSubscriptions ()
```

Returns an array of all the subjects that this client has subscribed to.

isSubscribed

```
boolean isSubscribed (String subject)
```

Returns `true` if this client has subscribed to messages with the specified subject.

publish

```
void publish (Message msg)
```

Sends a message to all users that have subscribed to the subject of this message.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

removeFilter

```
void removeFilter (MessageFilter filter)
```

Removes the specified filter.

sendToAll

```
void sendToAll (Message msg)
```

Sends a message to all users who are currently logged into the server.

sendToGroup

```
void sendToGroup (Message msg [, String groupName])
```

Sends a message to all the members of a particular group. If you omit the optional second parameter the message will be sent to the current group.

sendToServer

```
void sendToServer (Message msg)
```

Sends a message to the server. You must write a server side GroupExtension plugin that handles this message. Unlike all the other send methods, this one does not deliver the message to another client.

sendToUser

```
void sendToUser (Message msg, String username, boolean sameGroup)
```

Sends a message to one user. If the third parameter is true the message will only be delivered if the recipient is currently member of the same group.

subscribe

```
void subscribe (String subject)
```

Subscribes to all messages with the specified subject.

unsubscribe

```
void unsubscribe (String subject)
```

Unsubscribes from all messages with the specified subject.

unsubscribeAll

```
void unsubscribeAll ()
```

Unsubscribes from all messages.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured.

onMessage

```
onMessage (Message msg)
```

Event handler; invoked when a message is received that has no filter applied to.

onSubscribe

```
onSubscribe (String subject)
```

Event handler; invoked when messages with the specified subject have been subscribed.

onUnsubscribe

```
onUnsubscribe (String subject)
```

Event handler; invoked when messages with the specified subject have been unsubscribed.

onUnsubscribeAll

```
onUnsubscribeAll ()
```

Event handler; invoked when all messages have been unsubscribed.

PropertyLoader

The PropertyLoader object is only needed for properties of users who are not member of the current group or for properties of groups other than the current group. It can even be used for properties of users who are offline.

Example code to load two properties of a user might look as follows:

```
var pl = new omus.PropertyLoader();
var lis = new Object();
lis.onLoadUser = function (username,props) {
    trace("properties of user " + username + " were loaded");
    trace("age = " + props.age);
    trace("married = " + props.married);
};
lis.onError = function (error) {
    var name = error.getArguments()[0];
    trace("properties of user " + name + " could not be loaded");
    trace("error = " + error.getDescription());
};
pl.addListener(lis);
pl.loadUser("Jenny",["age","married"]);
```

Constructor Summary

new org.omus.**PropertyLoader** ()

Method Summary

void **addListener** (Object listener)
void **loadGroup** (String groupName, Array[String] propNames)
void **loadUser** (String username, Array[String] propNames)
void **removeListener** (Object listener)

Event Summary

onError (Error error)
onLoadGroup (String groupName, Object props)
onLoadUser (String username, Object props)

new

```
new org.omus.PropertyLoader ()
```

Creates a new PropertyLoader.

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the PropertyLoader object can generate.

loadGroup

```
void loadGroup (String groupName, Array[String] propNames)
```

Loads properties of a group.

loadUser

```
void loadUser (String username, Array[String] propNames)
```

Loads properties of a user.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured that was caused by one of the methods of the PropertyLoader object.

onLoadGroup

```
onLoadGroup (String groupName, Object props)
```

Event handler; invoked when properties of a group were loaded. The properties are included in a plain ActionScript object, not in a PropertySet object, because they cannot be modified or synchronized like those of the current group.

onLoadUser

```
onLoadUser (String username, Object props)
```

Event handler; invoked when properties of a user were loaded. The properties are included in a plain ActionScript object, not in a PropertySet object, because they cannot be modified or synchronized like those of users in the current group.

PropertySet

Each group and each user have a set of properties associated with them. The datatype as well as persistence and synchronization settings for each property must be defined in the XML configuration file on the server. See the Configuration chapter for details. The PropertySet object can be used to read or modify the values of individual properties, to load properties into the client or to synchronize them. You can obtain references to PropertySet objects with one of the following methods:

Properties of the current group	org.omus.group.getProperties()
Properties of the current user	org.omus.user.getProperties()
Properties of another user in the current group	org.omus.group.getUserProperties("username")

Properties of other groups or users who are not members of the current group are not loaded into the client automatically. Use the PropertyLoader object to read the values of these properties. Note that modified values will not be written to the database and not be reflected in other clients before you commit all changes with a call to `PropertySet.synchronize`. Example code to change three properties of a user named "Billy" and synchronize them might look as follows:

```
var propSet = org.omus.group.getUserProperties("Billy");
propSet.setValue("age",27);
propSet.setValue("married",false);
propSet.setValue("children",["Jenny","Carmel","Larry"]);
propSet.synchronize();
```

Method Summary

void **addListener** (Object listener)
boolean **contains** (String propName)
String|Group **getOwner** ()
ProperySet **getParent** ()
int **getPrimaryKey** ()
String **getType** (String propName)
any **getValue** (String propName)
boolean **isLoaded** (String propName)
boolean **isModified** (String propName)
boolean **isSynchronized** ()
boolean **isValid** ()
void **load** (Array[String] propNames)
void **loadAll** ()
void **removeListener** (Object listener)
void **setValue** (String propName, any newValue)
int **size** ()
void **synchronize** ()

Event Summary

onError (PropertySet subset, Error error)
onLoad (PropertySet subset)
onSynchronize (PropertySet subset, boolean clientRequest)

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the PropertySet object can generate.

contains

```
boolean contains (String propName)
```

Returns `true` if this PropertySet object contains a property with the specified name.

getOwner

```
String|Group getOwner ()
```

Returns the owner of this PropertySet object. If this object is associated with a user, the name of the user will be returned as a string. If it is associated with a group, a reference to the group object will be returned.

getParent

```
ProperySet getParent ()
```

Returns the parent object of this PropertySet or `null` if there is no parent. The PropertySet objects that get passed to the `onSynchronize` or `onLoad` event handlers are only subsets of the original PropertySet objects, containing only those properties that were affected by the synchronize or load operation. In these cases you can use this method to obtain the parent object which contains all properties.

getPrimaryKey

```
int getPrimaryKey ()
```

Returns the primary key for this PropertySet object, corresponding to the usrID and grpID fields in the userprops and groupprops tables in the database.

getType

```
String getType (String propName)
```

Returns the datatype of the property with the specified name.

getValue

```
any getValue (String propName)
```

Returns the value of the property with the specified name or `null` if the property was not loaded yet.

isLoaded

```
boolean isLoaded (String propName)
```

Returns `true` if the value of the property with the specified name has been loaded into the client.

isModified

```
boolean isModified (String propName)
```

Returns `true` if the property with the specified name was modified since the last `onSynchronize` event.

isSynchronized

```
boolean isSynchronized ()
```

Returns `true` if no property was modified since the last `onSynchronize` event.

isValid

```
boolean isValid ()
```

Returns `true` if this PropertySet object is associated with the current group or a user in the current group. If you store a reference to a PropertySet object in a variable and the user that owns this PropertySet left the group, the values of all properties in this set become stale because they will no longer be synchronized automatically.

load

```
void load (Array[String] propNames)
```

Loads all values of the properties with the specified names into the client. Please note that the values of some properties will be loaded into the client automatically depending on their configuration.

loadAll

```
void loadAll ()
```

Loads the values of all properties that are not loaded automatically.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

setValue

```
void setValue (String propName, any newValue)
```

Sets the value of the property with the specified name to the specified new value. The type of the new value must match the type you specified for this property in config.xml. Please note that the new values will not be written to the database and not be reflected in other clients before you commit all changes with a call to `PropertySet.synchronize`.

size

```
int size ()
```

Returns the number of properties in this set.

synchronize

```
void synchronize ()
```

Synchronizes all modified properties in this set according to the synchronization settings in the XML configuration file on the server. This may include writing the new values to the database or updating them in other clients of the current group depending on the configuration.

onError

```
onError (PropertySet subset, Error error)
```

Event handler; invoked when an error occured. The first argument is a PropertySet object that contains only the affected properties that could not be loaded or synchronized depending on the method that caused the error.

onLoad

```
onLoad (PropertySet subset)
```

Event handler; invoked when one or more properties were loaded. The argument is a PropertySet object that contains only the affected properties.

onSynchronize

```
onSynchronize (PropertySet subset, boolean clientRequest)
```

Event handler; invoked when one or more properties were synchronized. The first argument is a PropertySet object that contains only the affected properties. The second

argument is `true` if the `onSynchronize` event is the result of a call to `PropertySet.synchronize` in **this** client. It is `false` if the event was caused by another client or by the server.

Session

This object is used to connect to the server. The Session object is a singleton that will be created automatically. It can be referenced with `org.omus.session`.

Method Summary

void **addListener** (Object listener)
void **init** (String versionID, String address, int loginPort *[, int reconnectPort]*)
boolean **isConnected** ()
void **login** (String username, String password)
void **logout** ()
void **register** (String username, String password, String email)
void **removeListener** (Object listener)
boolean **setPasswordEncoder** (Function encoder)

Event Summary

onError (Error error)
onLogin (Object data)
onLogout (Error error)
onRegister (Object data)

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all events the Session object can generate.

init

```
void init (String versionID, String address, int loginPort [, int reconnectPort])
```

This method must be called before any attempt to login or register a user. It is best placed right after the `#include "oregano_as/init.as"` statement. `versionID` is an arbitrary string that must match the `versionID` attribute of the login node in config.xml on the server. It can be used to prevent that someone uses a deprecated version of your client application. `address` is either the human-readable name of your server (like "www.myServer.com") or the IP address. If Oregano Server runs on more than one machine, only the address of the Primary Server must be specified in the init method, the other servers will be found automatically. `loginPort` and `reconnectPort` must be 1024 or higher. `reconnectPort` is optional, it is only required if your server runs on more than one machine.

isConnected

```
boolean isConnected ()
```

Returns `true` if it is possible to send a command to the server. Returns `false` if the user is not logged in, if she is already logged out or the connection is broken. Furthermore it returns `false` if changing groups is in progress. You cannot send messages to the server before the connection to the new group has been fully established.

login

```
void login (String username, String password)
```

Used to login a user who is already registered with the server. All possible errors like invalid passwords are passed to the `onError` event.

logout

```
void logout ()
```

Closes the connection to the server.

register

```
void register (String username, String password, String email)
```

Used to register a new user with the server. All possible errors like duplicate user names are passed to the `onError` event.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

setPasswordEncoder

```
boolean setPasswordEncoder (Function encoder)
```

Optionally a function can be specified that encodes the password used in the login and register methods. It requires two arguments. The first argument is the password, the second is a string key send by the server that you can use to encode the password. The server then uses the same key for decoding. The function must return the encoded password. You have to write the corresponding server side decoder as a Java extension. See the section about the `org.omus.ext` package in the Server API chapter. A very simple example function might look as follows:

```
pwdEncoder = function (password, serverKey) {
    var offset = parseInt(serverKey);
    var encoded = "";
    for (var i = 0; i < password.length; i++) {
        var num = password.charCodeAt(i) + offset;
```

```
            encoded += num + "-";
        }
        return encoded;
    }
    org.omus.session.setPasswordEncoder(pwdEncoder);
```

onError

`onError (Error error)`

Event handler; invoked when an error occured. The error might be caused by a call to the `login` or `register` methods which might have failed. In addition the session object handles all the unknown errors which might be send by the server. In this case `error.getMethod()` returns `null`.

onLogin

`onLogin (Object data)`

Event handler; invoked when an attempt to login succeeded. The argument is an empty object unless you wrote a LoginExtension as a server side plugin which can be used to send additional data to the client.

onLogout

`onLogout (Error error)`

Event handler; invoked when the user has been logged out. You can use the specified error object to find out about the cause of the logout. See the section on error codes at the end of this chapter for a list of possible codes.

onRegister

`onRegister (Object data)`

Event handler; invoked when an attempt to register a user succeeded. The argument is an empty object unless you wrote a LoginExtension as a server side plugin which can be used to send additional data to the client.

Table

Table objects are the only datatype used for user and group properties that are not built-in ActionScript objects.

The rows of the table are plain ActionScript objects with their property names matching the column names of the table.

Note that the event handlers of the table object will only be invoked if the table is a value of a synchronized property. If the property is not synchronized or the table is

nested in another structured value in a property or you created a new table object by hand, the event handlers will not be invoked.

Table objects are especially useful, if you have large amounts of data in a user or group property, since the table object will only be partially synchronized. Only the modified, added or removed rows will be exchanged between clients, so you can save a lot of bandwith. Properties of type Array or Object, on the other hand, will always be fully synchronized.

Constructor Summary

new org.omus.**Table** (TableDefinition def)

Method Summary

void **addAllRows** (Table source)
void **addListener** (Object listener)
void **addRow** (Object row)
TableDefinition **getDefinition** ()
Object **getRow** (int index)
void **removeAllRows** ()
void **removeListener** (Object listener)
Object **removeRow** (int index)
int **size** ()
void **sort** (Function sortFunc)
void **sortBy** (String column, boolean ascending)
void **updateRow** (int index, Object row)

Event Summary

onAddRow (Table table, Object row, boolean clientRequest)
onRemoveAllRows (Table table, boolean clientRequest)
onRemoveRow (Table table, Object row, boolean clientRequest)
onUpdateRow (Table table, Object row, boolean clientRequest)

new

```
new org.omus.Table (TableDefinition def)
```

Creates a new table with the specified table definition. The TableDefinition object cannot be modified after it was used to create a table, but it can be used to create more than one table with the same definition.

addAllRows

```
void addAllRows (Table source)
```

Adds all rows of the specified table to this table. This method will only succeed if both tables have the same table definition.

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the Table object can generate.

addRow

```
void addRow (Object row)
```

Appends a single row to this table. This method will only succeed if the specified row object matches the table definition. See the entry for `TableDefinition.match` for details.

getDefinition

```
TableDefinition getDefinition ()
```

Returns the TableDefinition object for this table.

getRow

```
Object getRow (int index)
```

Returns the row at the specified index position.

removeAllRows

```
void removeAllRows ()
```

Removes all rows from this table object.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

removeRow

```
Object removeRow (int index)
```

Removes and returns the row at the specified index position.

size

```
int size ()
```

Returns the number of rows of this table.

sort

```
void sort (Function sortFunc)
```

Sorts the table using the specified comparison function. Given two arguments row1 and row2, the function should return the following values:

-1 if row1 should appear before row2
0 if row1 == row2
1 if row1 should appear after row2

sortBy

```
void sortBy (String column, boolean ascending)
```

Sorts the table by the column with the specified name.

updateRow

```
void updateRow (int index, Object row)
```

Updates the row at the specified index position. This method will only succeed if the specified row object matches the table definition. See the entry for `TableDefinition.match` for details.

onAddRow

```
onAddRow (Table table, Object row, boolean clientRequest)
```

Event handler; invoked when a new row was added to a synchronized table property. The second argument is the new row. The third argument is `true` if the `onAddRow` event is the result of a call to `Table.addRow` in this client. It is false if the event was caused by another client or by the server.

onRemoveAllRows

```
onRemoveAllRows (Table table, boolean clientRequest)
```

Event handler; invoked when all rows of a synchronized table property were removed. The second argument is `true` if the `onRemoveAllRows` event is the result of a call to `Table.removeAllRows` in this client. It is `false` if the event was caused by another client or by the server.

onRemoveRow

```
onRemoveRow (Table table, Object row, boolean clientRequest)
```

Event handler; invoked when a row of a synchronized table property was removed. The second argument is the row that was removed. The third argument is `true` if the `onRemoveRow` event is the result of a call to `Table.removeRow` in this client. It is `false` if the event was caused by another client or by the server.

onUpdateRow

```
onUpdateRow (Table table, Object row, boolean clientRequest)
```

Event handler; invoked when a row of a synchronized table property was updated. The second argument is the updated row. The third argument is `true` if the `onUpdateRow` event is the result of a call to `Table.updateRow` in this client. It is `false` if the event was caused by another client or by the server.

TableDefinition

TableDefinition objects are used to define the names and datatypes of the columns of a Table object.

Constructor Summary

new org.omus.**TableDefinition** ()

Method Summary

void **addColumn** (String name, String type)
int **getColumnCount** ()
String **getColumnName** (int index)
Array[String] **getColumnNames** ()
String **getColumnType** (int index)
boolean **matches** (Object row)

TableDefinition

```
new org.omus.TableDefinition ()
```

Creates a new empty TableDefinition object.

addColumn

```
void addColumn (String name, String type)
```

Adds a new column to this table definition. This method will fail if this TableDefinition object was already used to construct a new `org.omus.Table` object. The second argument must be one of the following strings: `boolean`, `int`, `long`, `float`, `string`, `date`, `array`, `object`, `table`.

getColumnCount

```
int getColumnCount ()
```

Returns the number of columns in this table definition.

getColumnName

```
String getColumnName (int index)
```

Returns the name of the column at the specified index position.

getColumnNames

```
Array[String] getColumnNames ()
```

Returns an array of all column names in the order they were added to this TableDefinition.

getColumnType

```
String getColumnType (int index)
```

Returns the datatype of the column at the specified index position.

matches

```
boolean matches (Object row)
```

Returns true if the specified object matches this table definition. The number of properties of the specified object must match the number of columns in this definition object and each property of the specified object must have a matching name and datatype in this definition object.

User

The user object represents the user that was logged in with this client. All other users who are currently member of the group are represented by their PropertySet objects instead.

The User object is a singleton that will be created automatically when a user logs in. It can be referenced with `org.omus.user`.

Method Summary

void **addListener** (Object listener)
Buddies **getBlacklist** ()
Buddies **getBuddies** ()
String **getEmail** ()
Mailbox **getInBox** ()
Locks **getLocks** ()
String **getName** ()
Mailbox **getOutBox** ()
String **getPassword** ()
int **getPermissions** ()
PropertySet **getProperties** ()

void **removeListener** (Object listener)
void **setEmail** (String newEmail)
void **setPassword** (String newPassword)
void **setPermissions** (int newPermissions)

Event Summary

onError (Error error)
onSetEmail ()
onSetPassword ()
onSetPermissions ()

addListener

```
void addListener (Object listener)
```

Adds the specified object as a listener for all the events the User object can generate.

getBlacklist

```
Buddies getBlacklist ()
```

Returns the Buddies object that represents the "blacklist" for this user. This method is new in Oregano 1.1.0. If a user adds another user to his blacklist he will not receive any Mail objects sent by that user. The blacklist returned by this method is an instance of the Buddies object, because it shares the same interface with the buddylist. See the section on the Buddies object for details.

getBuddies

```
Buddies getBuddies ()
```

Returns the Buddies object for this user. See the section on the Buddies object for details.

getEmail

```
String getEmail ()
```

Returns the email address of the user.

getInBox

```
Mailbox getInBox ()
```

Returns the Mailbox object representing the in box. See the section on the Mailbox object for details.

getLocks

```
Locks getLocks ()
```

Returns the Locks object for this user. See the section on the Locks object for details.

getName

```
String getName ()
```

Returns the name of the user.

getOutBox

```
Mailbox getOutBox ()
```

Returns the Mailbox object representing the out box. See the section on the Mailbox object for details.

getPassword

```
String getPassword ()
```

Returns the password of the user.

getPermissions

```
int getPermissions ()
```

Returns the permissions of the user, a number between 0 and 10:

0	user has been deleted
1	user has been banned
2-8	user has normal status
9-10	user has administrator status

The values between 2 and 8 make no difference to the Oregano Server. You can use them for your application logic. A newly registered user has a permission value set to 3.

A user who has administrator status will not be logged out if you switch the server to debug mode using the Flash Admin Client.

getProperties

```
PropertySet getProperties ()
```

Returns the PropertySet object associated with this user.

removeListener

```
void removeListener (Object listener)
```

Removes the specified listener from this object.

setEmail

```
void setEmail (String newEmail)
```

Sets the email address for this user. You must set the permissions for this method in config.xml on the server. See the Configuration chapter for details. This method may fail if the server is configured with `emailUnique="true"` and a user with the same email address is already registered.

setPassword

```
void setPassword (String newPassword)
```

Sets the password for this user. You must set the permissions for this method in config.xml on the server. See the Configuration chapter for details.

setPermissions

```
void setPermissions (int newPermissions)
```

Sets the password for this user. You must set the permissions for this method in config.xml on the server. See the Configuration chapter for details.

onError

```
onError (Error error)
```

Event handler; invoked when an error occured that was caused by one of the following methods of the user object: `setEmail`, `setPassword`, `setPermissions`.

onSetEmail

```
onSetEmail ()
```

Event handler; invoked when the email address was modified and written to the database.

onSetPassword

```
onSetPassword ()
```

Event handler; invoked when the password was modified and written to the database.

onSetPermissions

```
onSetPermissions ()
```

Event handler; invoked when the permission value was modified and written to the database.

Error Codes

Error descriptions:

ses-001	internal server error
ses-002	maximum number of connections exceeded
ses-003	login denied by custom server extension
ses-004	wrong client versionID
ses-005	server in debug mode or preparing for shutdown
ses-006	user already logged in
ses-007	user with given nick name already exists
ses-008	user is banned
ses-009	user with given email address already exists
ses-010	user with given email address is banned
ses-011	incorrect login
ses-012	no connection to server
msg-001	unknown message type
msg-002	error parsing message
msg-003	missing server extension for handling outgoing message
dbe-001	unable to connect to database
dbe-002	database error
dbe-003	SQL error
dbe-004	less than the required minimum of rows in the result set
dbe-005	less than the required minimum of affected rows
dbe-006	user is banned
dbe-007	user does not exist
dbe-008	group does not exist
dbe-009	missing <in>-parameter
dbe-010	missing <out>-parameter
dbe-011	type of transaction part does not match configuration
dbe-012	cannot handle fields set to SQL NULL
dbe-013	DbTransaction object was empty
dbe-014	execution of one or more statements denied
cgr-001	group does not exist
cgr-002	group cannot be created
cgr-003	group is closed
cgr-004	user limit of group reached
cgr-005	joining this group not permitted
grp-001	login group cannot be opened or closed
grp-003	client not owner of this lock
usr-001	changing email address not permitted
usr-002	changing password not permitted
usr-003	changing permissions not permitted
usr-004	illegal value for setting user permissions
mbx-001	mailbox limit exceeded
mbx.002	mail in out box cannot be marked as read
mbx.003	mail rejected: blacklist of recipient contains sender

bud-001	buddy list already contains this user
bud-002	buddy list did not contain this user
bud-003	limit of buddy list exceeded
prp-001	one or more properties do not exist
prp-002	datatype of one or more properties does not match configuration
prp-003	synchronization rejected by custom server extension
prp-004	PropertySet no longer valid
prp-005	changing groups failed
lgt-001	logout requested by client
lgt-002	logout requested by custom server extension
lgt-003	logout forced by administrator
lgt-004	user was banned
lgt-005	server switched to debug mode
lgt-006	server shutdown
lgt-007	connection broken

List of methods that can cause an onError event:

Method	Possible error codes
Buddies.insert	dbe-001, dbe-002, dbe-006, dbe-007, bud-001, bud-003
Buddies.load	dbe-001, dbe-002
Buddies.remove	dbe-001, dbe-002, bud-002
Buddies.removeAll	dbe-001, dbe-002
DbTransaction.execute	dbe-001, dbe-002, dbe-003, dbe-004, dbe-005, dbe-009, dbe-010, dbe-011, dbe-012, dbe-013, dbe-014
Group.change	dbe-001, dbe-002, cgr-001, cgr-002, cgr-003, cgr-004, cgr-005
Group.close	grp-001
Group.open	grp-001
Locks.release	grp-003
Mail.loadAttachment	dbe-001, dbe-002
Mail.mark	dbe-001, dbe-002, mbx-002
Mail.remove	dbe-001, dbe-002
Mail.send	dbe-001, dbe-002, dbe-006, dbe-007, mbx-001
Mailbox.load	dbe-001, dbe-002
Mailbox.removeAll	dbe-001, dbe-002
Messenger.sendToServer	msg-003
parsing incoming messages:	msg-001, msg-002
PropertyLoader.loadGroup	dbe-001, dbe-002, dbe-008, prp-001
PropertyLoader.loadUser	dbe-001, dbe-002, dbe-007, prp-001
PropertySet.load	dbe-001, dbe-002, prp-001, prp-004
PropertySet.loadAll	dbe-001, dbe-002, prp-004
PropertySet.synchronize	dbe-001,dbe-002, prp-002, prp-003, prp-004

Session.login	dbe-001, dbe-002, ses-002, ses-003, ses-004, ses-005, ses-006, ses-008, ses-011, ses-012
Session.register	dbe-001, dbe-002, ses-002, ses-003, ses-004, ses-005, ses-007, ses-008, ses-009, ses-010, ses-012
User.setEmail	dbe-001, dbe-002, usr-001
User.setPassword	dbe-001, dbe-002, usr-002
User.setPermissions	dbe-001, dbe-002, usr-003, usr-004

In addition each `onError` event can contain the error code `ses-001`.

CHAPTER 6
Server API

Writing server-side extensions

The extensive Client API enables you to develop a complex multiuser application without a single line of server-side code. But sometimes you need to move parts of your application logic to the server, especially if you add features that need to be synchronized within a group. Oregano Server comes with a set of interfaces that can be used to develop extensions in Java. The most important interface is GroupExtension which can be used to extend the functionality of a group.

There are four steps required to add an extension to the server:

1. Write a class that implements one of the five interfaces in package `org.omus.ext` (omus is the acronym for Oregano Multiuser Server)

2. Compile the class and move the .class file into the extensions directory inside the server runtime directory of Oregano Server.

3. Include the name of that class in config.xml. There are many places in this file where you can specify an extension class. It depends on the type of extension you wrote and on the way it is going to be used. See the Configuration chapter for details. If you wrote a class that implements GroupExtension you can specify it to be the default extension for all groups or to be used within a particular subset of groups only. In the latter case the entry in config.xml might look as follows:

```
<group
     configID="groundFloor"
     userLimit="50"
     extension="MyExtension"
     />
     <startup server="s1" name="kitchen" />
     <startup server="s1" name="bathroom" />
     <startup server="s1" name="entranceHall" />
</group>
```

   Now three groups named "kitchen", "bathroom" and "entranceHall" will be created when the server boots. All three groups will load and use the Java class MyExtension. Furthermore, all new groups created dynamically due to a client calling the `change` method of the Group object in the client API and specifying `"groundFloor"` as the configID, will use that class too.

4. Reboot Oregano Server.

Package Overview

org.omus.ext

Contains five interfaces that you can use to develop extensions for the Oregano Server. Each extension must implement one of these five interfaces and the corresponding class name must be specified in config.xml so that the server can load the extension. GroupExtension is the main interface of that package that can be used to extend the functionality of one or more groups.

org.omus.core

Contains the core classes. Group, User and PropertySet are classes that are very similar to their client-side twins of the same name. Use these objects to read and modify properties and to send messages to clients.

Furthermore this package contains a lot of additional utility classes.

org.omus.data

Contains the wrapper classes for the datatypes that can be exchanged with Flash Clients. Each property and each Message object contains data wrapped in one of the subclasses of the abstract superclass DataField.

The Message object used to send data to clients is included in this package too.

org.omus.db

Contains classes to read and write to and from the database. They are very similar to the corresponding objects in the Client API. Note that persistent properties will be written to the database automatically. You only need these classes if you created additional tables in the database.

Package org.omus.ext

Contains five interfaces that you can use to develop extensions for the Oregano Server. Each extension must implement one of these five interfaces and the corresponding class name must be specified in config.xml so that the server can load the extension.

GroupExtension

Interface

This is the main interface of the `org.omus.ext` package, you can use it to build custom groups (or "rooms"). It contains a set of methods that get invoked when a special event occured. All groups in the Oregano Server are single-threaded, so you do not need to care about synchronization issues. There is no need to create your own threads or Timer instances. If there is a time consuming task you can delegate it to a pool of worker threads with one of the methods of the TaskManager.

There are two places in config.xml where a GroupExtension can be specified. The defaultExtension attribute of the enclosing <groupConfig>-node and the extension attribute of each particular <group>-node. The former is useful if you want to add features to every group that will be created on the server, the latter is used if you write a "local" extension, only intended for groups created with a particular configID. If you combine both approaches, make sure that your local extension is a subclass of your defaultExtension, otherwise the functionality of your defaultExtension would be wiped out by the methods in your local one.

Let's assume you develop a default extension with the following class body:

```
class MyGlobalExtension implements org.omus.ext.GroupExtension { ... }
```

In this case the groupConfig node in config.xml should look as follows:

```
<groupConfig defaultExtension="MyGlobalExtension" ...
```

If you want to add special behaviour to a particular set of groups, you can define a subclass:

```
class MySpecialExtension extends MyGlobalExtension { ... }
```

Each method in this subclass should call the method in the superclass that it overrides if you want to combine the functionality of your local and global extensions.

The node for the groups that are supposed to load this extension might look as follows:

```
<group
    configID="spec"
    userLimit="50"
    extension="MySpecialExtension"
    >
    <startup server="serv1" name="spec_A" />
    <startup server="serv1" name="spec_B" />
</group>
```

When the server boots, two groups named "spec_A" and "spec_B" will be created and both will load the extension class MySpecialExtension which in turn is a subclass of MyGlobalExtension. Furthermore each group that will be created dynamically due to a client calling `org.omus.group.change("anyName","spec",false)` will load the same extension too.

Method Summary

void **finishGroupCreation** (Group group)
void **groupRemoved** ()
void **messageFromClient** (Message msg, User sender)
boolean **messageToGroup** (Message msg)
boolean **messageToUser** (Message msg, User recipient)
boolean **prepareGroupCreation** (GroupCreationData gcd)
boolean **syncGroupProperties** (PropertyUpdate newValues, User sender)
boolean **syncUserProperties** (PropertyUpdate newValues, User sender, User owner)
void **userJoined** (User user, DataRow extraData)
void **userLeft** (User user)

finishGroupCreation

```
void finishGroupCreation (Group group)
```

Invoked after the group has been created. Each class that implements this interface should keep a reference to the group object in a field.

groupRemoved

```
void groupRemoved ()
```

Invoked after the group has been removed. You can still modify the properties of this group which will be written to the database after this method finished executing.

messageFromClient

```
void messageFromClient (Message msg, User sender)
```

Invoked after a message from the specified sender has been received. This event originates from a client calling `sendToServer` in `org.omus.messenger`.

messageToGroup

```
boolean messageToGroup (Message msg)
```

Invoked before a message is sent to all the clients who are currently member of this group. This event may originate from a client calling `sendToGroup` in `org.omus.messenger` or from any server extension calling `sendToGroup` in the MessagingManager. Return true if you want the message to proceed on its way to the clients.

messageToUser

```
boolean messageToUser (Message msg, User recipient)
```

Invoked before a message is sent to the specified user. This event may originate from a client calling `sendToUser` or `sendToAll` in `org.omus.messenger` or from any server extension calling `sendToGroup` or `sendToAll` in the MessagingManager. Return true if you want the message to proceed on its way to the client.

prepareGroupCreation

```
boolean prepareGroupCreation (GroupCreationData gcd)
```

Invoked before the group will be created. The GroupCreationData object contains methods to read the name and the configID of the group that is supposed to be created. If this method returns false, the creation of this group will be aborted. This way you can implement some kind of error checking on the server or load some additional data before the group is created. You can use the `setErrorCode` method of the GroupCreationData object to send customized error codes to the clients in case you want to prevent the creation of that group. Otherwise a default error code will be sent.

syncGroupProperties

```
boolean syncGroupProperties (PropertyUpdate newValues, User sender)
```

Invoked before the properties of this group will be synchronized. The sender represents the client that modified the properties. PropertyUpdate is a subclass of PropertySet. This object does not contain all the properties that are associated with this group, but only the modified ones. You can implement some kind of error checking and only return true if you want the new values to be accepted and synchronized. If you return false to reset all affected properties to their old value, you can use the `setErrorCode` method of the PropertyUpdate object to send customized error codes to the client who attempted to change the value. Otherwise a default error code will be sent.

syncUserProperties

```
boolean syncUserProperties (PropertyUpdate newValues, User sender, User owner)
```

Invoked before the properties of the specified owner will be synchronized. The sender represents the client that modified the properties. PropertyUpdate is a subclass of PropertySet. This object does not contain all the properties that are associated with the owner, but only the modified ones. You can implement some kind of error checking and only return true if you want the new values to be accepted and synchronized. If you return false to reset all affected properties to their old value, you can use the `setErrorCode` method of the PropertyUpdate object to send customized error codes to the client who attempted to change the value. Otherwise a default error code will be sent.

userJoined

```
void userJoined (User user, DataRow extraData)
```

Invoked after a user has joined the group. You can use the DataRow object, which is empty initially, to send additional data to the client which will be received in the parameter of the `onChange` event handler of the group object in the client. (if you develop an extension to the login group, the additional data will be receceived in the `onLogin` or `onRegister` events.

userLeft

```
void userLeft (User user)
```

Invoked after a user has left the group. You can still change the properties of the user. This method gets invoked before the `userJoined` method of the new group will be invoked.

KeyGenerator

Interface

A key generated by a class that implements this interface can be used to encode passwords on the client. This extension must be used together with a PasswordDecoder extension. Furthermore the corresponding encoding function must be written in

ActionScript and set with the `setPasswordEncoder` method in `org.omus.session` in the client. The generated key will be passed to the encoding method in the client and to the decoding method on the server.

You must specify the name of the class that implements this interface in the `<extensions>`-node in config.xml.

Method Summary

String **generateKey** ()

generateKey

```
String generateKey ()
```

Invoked before a client will be requested to send the username and password; this method must return a key that will be passed to the encoding method in the client and to the decoding method on the server.

LoginExtension

Interface

This interface is useful if you want to modify the way users are registered or logged in. You must specify the name of the class that implements this interface in the <extensions>-node in config.xml.

Method Summary

void **finishLogin** (User user)
void **finishRegister** (User user)
boolean **prepareLogin** (AuthenticationData data)
boolean **prepareRegister** (RegistrationData data)

finishLogin

```
void finishLogin (User user)
```

Invoked after the user has been logged in.

finishRegister

```
void finishRegister (User user)
```

Invoked after the user has been registered. If the password for the new user was generated by this LoginExtension, this is the right place to send an email to the new user containing a registration confirmation and her password.

prepareLogin

```
boolean prepareLogin (AuthenticationData data, InetAddess clientAddress,
                      int clientPort)
```

Invoked before a registered user is logged in. The AuthenticationData object contains methods to read or set the name and password of the user. If this method returns false, the login will be aborted. This way you can implement some kind of error checking on the server or load some additional data before the user will be logged in. You can even use this method to authenticate the user with a different server application if Oregano Server is tied to a legacy system. You can use the `setErrorCode` method of the AuthenticationData object to send customized error codes to the clients. Otherwise a default error code will be sent.

prepareRegister

```
boolean prepareRegister (RegistrationData data, InetAddess clientAddress,
                         int clientPort)
```

Invoked before a new user is registered. The RegistrationData object contains methods to read or set the name, password or email address of the user. If this method returns false, the registration will be aborted. This way you can implement some kind of error checking on the server or load some additional data before the user will be registered. It is a good idea to generate the password on the server and not to use the one sent from the client. You can use the `finishRegister` method of this extension to send the generated password in an email to the new user and thus verify the email address that she entered. To send customized error codes to the clients, you can use the `setErrorCode` method of the AuthenticationData object. Otherwise a default error code will be sent.

LogoutExtension

Interface

A logout extension can be written to modify the properties of the user before they are written to the database. You must specify the name of the class that implements this interface in the `<extensions>`-node in config.xml.

Method Summary

void **prepareLogout** (User user)

prepareLogout

```
void prepareLogout (User user)
```

Invoked when the specified user will be logged out. You can still modify the properties of this user which will be written to the database after this method executed.

PasswordDecoder

Interface

Provides a method to decode an encrypted password. This extension must be used together with a KeyGenerator extension. You must specify the name of the class that implements this interface in the `<extensions>`-node in config.xml.

Method Summary

String **decode** (String password, String secretKey)

decode

```
String decode (String password, String secretKey)
```

Invoked when a user will be logged in, the specified password was encoded by the client; this method must return the decoded password using the specified key which is the same that was used by the client to encode the password.

Package org.omus.core

Contains the core classes of the Oregano Server. Group, User and PropertySet are classes that are very similar to their client-side twins of the same name. Use these objects to read and modify properties and to send messages to clients.

Furthermore this package contains a lot of additional utility classes.

AuthenticationData

Contains authentication data of a user. Gets passed to the `prepareLogin` method in `org.omus.ext.LoginExtension`.

Method Summary

String **getErrorCode** ()
String **getPassword** ()
String **getUsername** ()
void **setErrorCode** (String newCode)
void **setPassword** (String newPassword)
void **setUsername** (String newName)

getErrorCode

```
String getErrorCode ()
```

Returns the error code or null if none was specified yet.

getPassword

```
String getPassword ()
```

Returns the password of the user.

getUsername

```
String getUsername ()
```

Returns the name of the user.

setErrorCode

```
void setErrorCode (String newCode)
```

Sets the error code to the specified string. The code will be prefixed with "cld-" automatically to avoid confusion with built-in error codes. It will be sent to the client if the `prepareLogin` method in `org.omus.ext.LoginExtension` returns false so that the login will be aborted.

setPassword

```
void setPassword (String newPassword)
```

Sets the password of the user to the specified string.

setUsername

```
void setUsername (String newName)
```

Sets the name of the user to the specified string.

Command

Interface

An interface for commands that will be passed to one of the methods in TaskManager, to `addCommand` in Group or to `setUndeliveredCommand` in Message.

Method Summary

void **execute** ()

execute

```
void execute ()
```

Executes the command.

Group

Represents a group of users. For each group that will be created due to an entry in config.xml or dynamically by a client, an instance of Group will be created. The only way to obtain a reference to a Group instance is the `groupCreated` method in `org.omus.ext.GroupExtension`. The Java Group class is similar to the ActionScript Group object in the Client API.

Method Summary

boolean **addCommand** (Command com)
void **close** ()
Iterator **getAllUsers** ()
String **getConfigID** ()
String **getName** ()
PropertySet **getProperties** ()
User **getUser** (String name)
int **getUserCount** ()
int **getUserLimit** ()
boolean **isClosed** ()
boolean **isFull** ()
void **open** ()
void **sendToClients** (Message msg)
void **sendToClients** (Message msg, User exclude)

addCommand

```
boolean addCommand (Command com)
```

Adds the specified command to the queue of this group for later execution. You will rarely need this method because all of the methods that are invoked in your GroupExtension get executed in the group queue anyway. All groups in the Oregano Server are single-threaded. Many methods of the Group, User, and PropertySet classes are protected against access from other threads because they are not synchronized. If there is a time consuming task that you delegated to a worker thread with one of the methods of the TaskManager, you cannot call these methods from within that task. So if you use the TaskManager to read from the database, for example, and want to use the values that you read to modify the properties of a user, you have to write a command that performs this task and put it into the group queue. Example code migth look like this:

```
final String result = null;
Command com1 = new Command () {
    public void execute () {
        result = aTimeConsumingCalculation();
        Command com2 = new Command () {
            public void execute () {
                PropertySet ps = group.getProperties();
                StringField st = (StringField)ps.getValue("foo");
                st.setString(result);
```

```
                }
            };
            group.addCommand(com2);
        }
    };
    Services.getTaskManager.executeNow(com1);
```

With this approach you avoid to slow down the group queue with your time consuming calculation, but are still able to pass its result to one of the protected methods.

close

```
void close ()
```

Closes the group so that no other users can join.

getAllUsers

```
Iterator getAllUsers ()
```

Returns an Iterator over all the User instances of this group.

getConfigID

```
String getConfigID ()
```

Returns the configID of the group. It identifies the corresponding `<group>`-node in config.xml that was used to configure this group. You can create an unlimited number of groups with the same configID but different names. See the Configuration chapter for more details.

getName

```
String getName ()
```

Returns the name of the group.

getProperties

```
PropertySet getProperties ()
```

Returns the PropertySet object associated with this group.

getUser

```
User getUser (String name)
```

Returns the User object identified by the specified user name.

getUserCount

```
int getUserCount ()
```

Returns the number of users who are currently member of this group.

getUserLimit

```
int getUserLimit ()
```

Returns the maximum number of users permitted for this group.

isClosed

```
boolean isClosed ()
```

Returns true if the group is closed so that no other users can join.

isFull

```
boolean isFull ()
```

Returns true if the maximum number of users has been reached.

open

```
void open ()
```

Opens the group for other users to join.

sendToClients

```
void sendToClients (Message msg)
```

Sends the specified Message to all the clients in this group. The message will be received in the `onMessage` event handler in the client side Messenger object.

sendToClients

```
void sendToClients (Message msg, User exclude)
```

Sends the specified Message to all the clients in this group except for the specified user. The message will be received in the `onMessage` event handler in the client side Messenger object.

GroupCreationData

Contains information about a group that is going to be created. Gets passed to the `prepareGroupCreation` method in `org.omus.ext.GroupExtension`.

Method Summary

String **getConfigID** ()
String **getErrorCode** ()
String **getName** ()
void **setErrorCode** (String newCode)

getConfigID

```
String getConfigID ()
```

Returns the configID of the group. It identifies the `<group>`-node in config.xml that was used to configure this group. You can have an unlimited number of groups with the same configID but different names. See the Configuration chapter for more details.

getErrorCode

```
String getErrorCode ()
```

Returns the error code or null if none was specified yet.

getName

```
String getName ()
```

Returns the name of the group.

setErrorCode

```
void setErrorCode (String newCode)
```

Sets the error code to the specified string. The code will be prefixed with "cld-" automatically to avoid confusion with built-in error codes. It will be sent to the client if the `prepareGroupCreation` method in `org.omus.ext.GroupExtension` returns false so that the creation of the group will be aborted.

LogManager

Contains methods to create custom log entries in the logging table on the server. The object uses four different log-levels: `DEBUG`, `INFO`, `WARN` and `ERROR`. A log-level is an indication of the importance of a message. Depending on the mininum log-level you specified for the `<extensions>`-node in the logging section in config.xml a log entry might be ignored or written to the database.

If the configuration looks like this

```
<extensions  level="warn" />
```

all log entries of level WARN or higher will be written to the database (or to a file, depending on the configuration). Accordingly calling one of the debug or info methods in LogManager will be ignored in this case.

Method Summary

void **debug** (String errorCode)
void **debug** (String errorCode, String info)
void **debug** (String errorCode, Exception e)
void **debug** (String errorCode, Exception e, String info)
boolean **debugEnabled** ()
void **error** (String errorCode)
void **error** (String errorCode, String info)
void **error** (String errorCode, Exception e)
void **error** (String errorCode, Exception e, String info)
boolean **errorEnabled** ()
void **info** (String errorCode)
void **info** (String errorCode, String info)
void **info** (String errorCode, Exception e)
void **info** (String errorCode, Exception e, String info)
boolean **infoEnabled** ()
void **warn** (String errorCode)
void **warn** (String errorCode, String info)
void **warn** (String errorCode, Exception e)
void **warn** (String errorCode, Exception e, String info)
boolean **warnEnabled** ()

debug

```
void debug (String errorCode)
```

Creates a log entry with the specified error code if the LogManager is configured to process messages of level DEBUG or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

debug

```
void debug (String errorCode, String info)
```

Creates a log entry with the specified error code and info string if the LogManager is configured to process messages of level DEBUG or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

debug

```
void debug (String errorCode, Exception e)
```

Creates a log entry with the specified error code and the stackTrace of the specified Exception if the LogManager is configured to process messages of level DEBUG or higher.

The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

debug

```
void debug (String errorCode, Exception e, String info)
```

Creates a log entry with the specified error code, info string and the stackTrace of the specified Exception, if the LogManager is configured to process messages of level DEBUG or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

debugEnabled

```
boolean debugEnabled ()
```

Returns true if the LogManager is configured to process log entries of level DEBUG or higher.

error

```
void error (String errorCode)
```

Creates a log entry with the specified error code if the LogManager is configured to process messages of level ERROR. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

error

```
void error (String errorCode, String info)
```

Creates a log entry with the specified error code and info string if the LogManager is configured to process messages of level ERROR. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

error

```
void error (String errorCode, Exception e)
```

Creates a log entry with the specified error code and the stackTrace of the specified Exception, if the LogManager is configured to process messages of level ERROR. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

error

```
void error (String errorCode, Exception e, String info)
```

Creates a log entry with the specified error code, info string and the stackTrace of the specified Exception, if the LogManager is configured to process messages of level

ERROR. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

errorEnabled

```
boolean errorEnabled ()
```

Returns true if the LogManager is configured to process log entries of level ERROR or higher.

info

```
void info (String errorCode)
```

Creates a log entry with the specified error code if the LogManager is configured to process messages of level INFO or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

info

```
void info (String errorCode, String info)
```

Creates a log entry with the specified error code and info string if the LogManager is configured to process messages of level INFO or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

info

```
void info (String errorCode, Exception e)
```

Creates a log entry with the specified error code and the stackTrace of the specified Exception, if the LogManager is configured to process messages of level INFO or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

info

```
void info (String errorCode, Exception e, String info)
```

Creates a log entry with the specified error code, info string and the stackTrace of the specified Exception, if the LogManager is configured to process messages of level INFO or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

infoEnabled

```
boolean infoEnabled ()
```

Returns true if the LogManager is configured to process log entries of level INFO or higher.

warn

```
void warn (String errorCode)
```

Creates a log entry with the specified error code if the LogManager is configured to process messages of level `WARN` or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

warn

```
void warn (String errorCode, String info)
```

Creates a log entry with the specified error code and info string if the LogManager is configured to process messages of level `WARN` or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

warn

```
void warn (String errorCode, Exception e)
```

Creates a log entry with the specified error code and the stackTrace of the specified Exception, if the LogManager is configured to process messages of level `WARN` or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

warn

```
void warn (String errorCode, Exception e, String info)
```

Creates a log entry with the specified error code, info string and the stackTrace of the specified Exception, if the LogManager is configured to process messages of level `WARN` or higher. The error code will be automatically prefixed with "ext-" to avoid confusion with built-in error codes.

warnEnabled

```
boolean warnEnabled ()
```

Returns true if the LogManager is configured to process log entries of level `WARN` or higher.

MessagingManager

Handles the delivery of messages. The MessagingManager is intended for messages aimed at targets outside the current group. For messages that you want to send to clients within the current group use `sentToClients` in Group or `sendToClient` in User instead. The MessagingManager is similar to the ActionScript Messenger object in the Client API.

Method Summary

void **publish** (Message msg)
void **sendToAll** (Message msg)
void **sendToGroup** (Message msg, String groupName)
void **sendToUser** (Message msg, String recipient)
void **subscribe** (User user, String subject)
void **unsubscribe** (User user, String subject)
void **unsubscribeAll** (User user)

publish

```
void publish (Message msg)
```

Sends a message to all users that have subscribed to the subject of this message. It will be passed to the `messageToUser` method in `org.omus.ext.GroupExtension` for all the recipients and then forwarded to the clients and received in the `onMessage` event handler in the client side Messenger object.

sendToAll

```
void sendToAll (Message msg)
```

Sends a message to all users who are currently logged into the server. It will be passed to the `messageToUser` method in `org.omus.ext.GroupExtension` for all the recipients and then forwarded to the clients and received in the `onMessage` event handler in the client side Messenger object.

sendToGroup

```
void sendToGroup (Message msg, String groupName)
```

Sends a message to all the members of a particular group. It will be passed to the `messageToGroup` method in `org.omus.ext.GroupExtension` in the target group and then forwarded to the client and received in the `onMessage` event handler in the client side Messenger object.

sendToUser

```
void sendToUser (Message msg, String recipient)
```

Sends a message to one user. It will be passed to the `messageToUser` method in `org.omus.ext.GroupExtension` in the group that the user is currently member of and then forwarded to the client and received in the `onMessage` event handler in the client side Messenger object.

subscribe

```
void subscribe (User user, String subject)
```

Subscribes the specified user to all messages with the specified subject.

unsubscribe

```
void unsubscribe (User user, String subject)
```

Unsubscribes the specified user from all messages with the specified subject.

unsubscribeAll

```
void unsubscribeAll (User user)
```

Unsubscribes the specified user from all messages.

PropertySet

implements Serializable

Represents a collection of properties. The PropertySet class is similar to the ActionScript PropertySet object in the Client API. Each group and each user have a set of properties associated with them. The datatype as well as persistence and synchronization settings for each property must be defined in the XML configuration file on the server. See the Configuration chapter for details. The PropertySet object can be used to read individual property values, to load properties from the database or to synchronize them. The actual modification of the value of a property is carried out through methods of the DataField instance that represents the property. Please note that modified values will not be written to the database and not be reflected in the clients before you commit all changes with a call to `synchronize`. Example code to change some properties of a user and synchronize them might look as follows:

```
PropertySet ps = group.getProperties();
IntField age = (IntField)ps.getValue("age");
age.setInt(27);
BooleanField married = (BooleanField)ps.getValue("married");
married.setBoolean(false);
try {
    ps.synchronize();
} catch (org.omus.db.DbException dbe) {
    System.out.println("ERROR: " + dbe);
}
```

Method Summary

boolean **contains** (String name)
Object **getOwner** ()
IntField **getPrimaryKey** ()
Class **getType** (String name)
DataField **getValue** (String name)
boolean **isLoaded** (String name)
boolean **isModified** (String name)
void **load** (String[] propNames)
void **loadAll** ()
int **size** ()
void **synchronize** ()

contains

```
boolean contains (String name)
```

Returns true if this PropertySet object contains a property with the specified name.

getOwner

```
Object getOwner ()
```

Returns the owner of this PropertySet object. The returned object is either an instance of Group or an instance of User.

getPrimaryKey

```
IntField getPrimaryKey ()
```

Returns the primary key for this PropertySet object corresponding to the usrID and grpID fields in the userprops and groupprops tables in the database.

getType

```
Class getType (String name)
```

Returns the type of the property with the specified name.

getValue

```
DataField getValue (String name)
```

Returns the value of the property with the specified name or null if the property was not loaded yet. To modify the value use one of the methods of the returned DataField.

isLoaded

```
boolean isLoaded (String name)
```

Returns true if the value of the property with the specified name has been loaded from the database.

isModified

```
boolean isModified (String name)
```

Returns true if the property with the specified name was modified since the last synchronization.

load

```
void load (String[] propNames)
```

Loads all values of the properties with the specified names into the client. Note that the values of all properties except for those with a cacheLevel attribute set to "`off`" will be loaded into the server-side PropertySet object automatically. See the Configuration chapter for more details.

loadAll

```
void loadAll ()
```

Loads the values of all properties that are not loaded automatically.

size

```
int size ()
```

Returns the number of properties in this set.

synchronize

```
void synchronize ()
```

Synchronizes all modified properties in this set according to the synchronization settings in the XML configuration file on the server. This may include writing the new values to the database or updating them in one or all clients of the current group depending on the configuration.

PropertyUpdate

extends PropertySet
implements Serializable

A subset of a PropertySet instance that gets passed to the `syncUserProperties` and `syncGroupProperties` methods in `org.omus.ext.GroupExtension`. It contains only the properties that were modified. It inherits all the methods of PropertySet and adds three additional methods to read and set an additional error code or to obtain a reference to the parent ProprtySet.

Method Summary

String **getErrorCode** ()
ProperySet **getParent** ()
void **setErrorCode** (String err)

getErrorCode

```
String getErrorCode ()
```

Returns the error code of this instance or null if none was specified yet.

getParent

```
ProperySet getParent ()
```

Returns the parent object of this PropertyUpdate. The PropertyUpdate instances that get passed to the `syncUserProperties` or `syncGroupProperties` methods in the GroupExtension class are only subsets of the original PropertySet objects, containing only those properties that were modified. In these cases you can use this method to obtain the parent object which contains all properties.

setErrorCode

```
void setErrorCode (String err)
```

Sets the error code of this instance to the specified string. Useful if you want to abort a synchronization from within the `syncUserProperties` and `syncGroupProperties` methods in `org.omus.ext.GroupExtension` and send a customized error code to the client. Otherwise a default error code would be sent.

RegistrationData

extends AuthenticationData

Contains registration data of a user. Gets passed to the `prepareRegister` method in `org.omus.ext.LoginExtension`. It inherits all the methods of Authentication data. It adds two methods to handle the email address of the user.

Method Summary

String **getEmail** ()
void **setEmail** (String newMail)

getEmail

```
String getEmail ()
```

Returns the email address of the user.

setEmail

```
void setEmail (String newMail)
```

Sets the email address of the user to the specified string.

Services

A static utility class to retrieve instances of one of the Manager classes.

Method Summary

static LogManager **getLogManager** ()
static MessagingManager **getMessagingManager** ()
static TaskManager **getTaskManager** ()

getMessagingManager

```
static MessagingManager getMessagingManager ()
```

Returns the MessagingManager.

getTaskManager

```
static TaskManager getTaskManager ()
```

Returns the TaskManager.

getLogManager

```
static LogManager getLogManager ()
```

Returns the LogManager.

TaskManager

A facility to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals. Some of the methods of TaskManager simply delegate the specified TimerTask to the encapsulated Timer instance. Others require a Command object to be specified and are an extension to the functionality found in java.util.Timer.

You should avoid to create your own threads or Timer instances, it may prevent the correct shutdown of the Oregano Server.

Method Summary

void **cancel** (Command com)
void **executeDaily** (Command com)
void **executeMonthly** (Command com, int dayInMonth)
void **executeNow** (Command com)
void **executeWeekly** (Command com, int dayInWeek)
void **schedule** (TimerTask task, Date time)
void **schedule** (TimerTask task, Date firstTime, long period)
void **schedule** (TimerTask task, long delay)
void **schedule** (TimerTask task, long delay, long period)
void **scheduleAtFixedRate** (TimerTask task, Date firstTime, long period)
void **scheduleAtFixedRate** (TimerTask task, long delay, long period)

cancel

```
void cancel (Command com)
```

Cancels regular execution of the specified Command.

executeDaily

```
void executeDaily (Command com)
```

Executes the specified Command every day at the time specified in the `<statistics>`-node in config.xml.

executeMonthly

```
void executeMonthly (Command com, int dayInMonth)
```

Executes the specified Command each month on the specified day at the time specified in the `<statistics>`-node in config.xml. Valid values for dayInMonth range from 1 to 31 - obviously.

executeNow

```
void executeNow (Command com)
```

Delegates the execution of the specified Command to one of the worker threads in the Oregano Thread pool.

executeWeekly

```
void executeWeekly (Command com, int dayInWeek)
```

Executes the specified Command once a week on the specified day at the time specified in the <statistics>-node in config.xml. Valid values for dayInWeek range from 1 (Sunday) to 7 (Saturday).

schedule

```
void schedule (TimerTask task, Date time)
```

Schedules the specified task for execution at the specified time.

schedule

```
void schedule (TimerTask task, Date firstTime, long period)
```

Schedules the specified task for repeated fixed-delay execution, beginning at the specified time. For details see the documentation for the corresponding method in java.util.Timer.

schedule

```
void schedule (TimerTask task, long delay)
```

Schedules the specified task for execution after the specified delay.

schedule

```
void schedule (TimerTask task, long delay, long period)
```

Schedules the specified task for repeated fixed-delay execution, beginning after the specified delay. For details see the documentation for the corresponding method in java.util.Timer.

scheduleAtFixedRate

```
void scheduleAtFixedRate (TimerTask task, Date firstTime, long period)
```

Schedules the specified task for repeated fixed-rate execution, beginning at the specified time. For details see the documentation for the corresponding method in java.util.Timer.

scheduleAtFixedRate

```
void scheduleAtFixedRate (TimerTask task, long delay, long period)
```

Schedules the specified task for repeated fixed-rate execution, beginning after the specified delay. For details see the documentation for the corresponding method in java.util.Timer.

User

Represents a single user.

Method Summary

String	**getEmail** ()
String	**getName** ()
String	**getPassword** ()
int	**getPermissions** ()
PropertySet	**getProperties** ()
void	**logout** (String errorCode)
void	**sendToClient** (Message msg)

getEmail

```
String getEmail ()
```

Returns the email address of the user.

getName

```
String getName ()
```

Returns the name of the user.

getPassword

```
String getPassword ()
```

Returns the password of the user.

getPermissions

```
int getPermissions ()
```

Returns the permissions of the user, a number between 0 and 10.

0	user has been deleted
1	user has been banned
2-8	user has normal status
9-10	user has administrator status

The values between 2 and 8 make no difference to the Oregano Server. You can use them for your application logic. A user who has administrator status will not be logged out if you switch the server to debug mode using the admin client.

getProperties

```
PropertySet getProperties ()
```

Returns the PropertySet object associated with this user.

logout

```
void logout (String errorCode)
```

Logs out the user, passing the specified error code to the client before closing the connection.

sendToClient

```
void sendToClient (Message msg)
```

Sends the specified Message to the Flash client of this user. The message will be received in the `onMessage` event handler in the client side Messenger object.

Package org.omus.data

Contains classes that represent ActionScript datatypes that can be passed to the server and written to the database. All the conversion steps are done automatically. ActionScript datatypes will be marshalled, sent to the server, converted to their corresponding Java types and - if they are part of a persistent property or mail attachment - written to a field in the database with a matching SQL type, according to the following conversion table:

ActionScript datatype	Java datatype	SQL type
boolean	BooleanField	TINYINT
number	CounterField	INTEGER
number	IntField	INTEGER
number	LongField	BIGINT
number	DoubleField	DOUBLE
Date	DateField	TIMESTAMP
String	StringField	VARCHAR/LONGVARCHAR
Array	DataList	VARCHAR/LONGVARCHAR
Object	DataRow	VARCHAR/LONGVARCHAR
org.omus.Table	DataTable	VARCHAR/LONGVARCHAR

All classes in the Java column are subclasses of DataField. At first glance you might think that working with those Oregano Wrapper types like IntField is cumbersome compared to working with Java primitive datatypes or classes like ArrayList or HashMap. But the DataField family is efficient for two reasons:

First they keep track of all changes. Each DataField type generates `fieldChange` events and you can register listener objects to handle this event. In the case of complex datatypes like DataList, the event bubbles up in the container hierarchy until it reaches the outermost container. The PropertySet objects make use of this mechanism to synchronize only the properties that were modified.

The second advantage is, that each DataField object caches its marshalled data, that gets send to Flash clients. Thus, if two clients want to load the same property and it has not been modified between these two requests, the cached value will be sent to the second client, so that there is no performance penalty for remarshalling the value.

BooleanField

extends DataField
implements Cloneable, Serializable

The class corresponding to the boolean datatype in ActionScript.

Constructor Summary

BooleanField (String name, boolean value)

Method Summary

boolean **getBoolean** ()
void **setBoolean** (boolean newValue)

BooleanField

```
BooleanField (String name, boolean value)
```

Creates a new BooleanField with the specified name and value.

getBoolean

```
boolean getBoolean ()
```

Returns the value of this BooleanField object as a boolean primitive.

setBoolean

```
void setBoolean (boolean newValue)
```

Sets the value of this BooleanField to the specified boolean primitive.

CounterField

extends DataField
implements Cloneable, Serializable

A special class for synchronizing int values. This is the only subclass of DataField that is not just a Java class corresponding to an ActionScript datatype. It makes it possible to modify int values relative to their current value and thus avoid any synchronization issues that might occur if different clients modify the int value of a group property concurrently.

An example: Let's assume you created a Flash client that shows a basket and each user in the group can put any amount of apples into the basket and take any amount out of it. If two users put an apple into that basket concurrrently and both clients add 1 to the current number of apples and then send the new total value to the server, only one of the two apples gets added to the group property on the server side. If a property is configured to be a counter, the `setValue` method of the client side PropertySet object changes the value relative to the current value. This way the server receives two "add 1 apple" commands and thus both apples are added to the basket. In other words: If `"foo"` is the name of a property configured as a counter, then `setValue("foo",-4)` called in PropertySet in the client means: "subtract 4 from the current value" and only the relative change is submitted to the server.

Constructor Summary

CounterField (String name, int initialValue)

Method Summary

void **add** (int dif)
int **get** ()
void **reset** ()
void **set** (int newValue)

CounterField

```
CounterField (String name, int initialValue)
```

Creates a new CounterField with the specified name and initial value.

add

```
void add (int dif)
```

Adds the specified value to the int value of this CounterField. Negative values are permitted.

get

```
int get ()
```

Returns the current value of this CounterField.

reset

```
void reset ()
```

Resets the value of this CounterField to the initial value.

set

```
void set (int newValue)
```

Sets the value of this CounterField to the specified value.

DataField

implements Cloneable, Serializable

The abstract superclass of all DataFields.

Method Summary

void **accept** (FieldVisitor fv)
void **addListener** (FieldListener fl)
Object **clone** ()

String **getName** ()
void **removeListener** (FieldListener fl)

accept

```
void accept (FieldVisitor fv)
```

Accepts a visitor. The visitor pattern is useful if you want to extend the functionality of all DataFields without modifying their API.

addListener

```
void addListener (FieldListener fl)
```

Adds a listener to this DataField that gets notified when the value of this object was modified.

clone

```
Object clone ()
```

Returns a clone of this DataField

getName

```
String getName ()
```

Returns the name of this DataField.

removeListener

```
void removeListener (FieldListener fl)
```

Removes the specified listener from this DataField.

DataList

extends DataField
implements Cloneable, Serializable

The class corresponding to the Array object in ActionScript. When an array is sent to to the server, it will be converted to a DataList instance and each element of the array is converted to the corresponding DataField.

Constructor Summary

DataList (String name)
DataList (String name, DataList source)

Method Summary

void **add** (DataField df)
void **add** (int index, DataField df)
void **addAll** (DataList list)
void **addAll** (int index, DataList list)
Object **clone** ()
DataField **get** (int index)
Iterator **iterator** ()
DataField **remove** (int index)
void **removeAll** ()
int **size** ()
void **sort** (DataList.Comparator listComp)

DataList

```
DataList (String name)
```

Creates a new DataList with the specified name.

DataList

```
DataList (String name, DataList dl)
```

Creates a new DataList with the specified name and the same DataFields as the specified DataList.

add

```
void add (DataField df)
```

Appends the specified DataField to the end of this list.

add

```
void add (int index, DataField df)
```

Inserts the specified DataField at the specified position in this list.

addAll

```
void addAll (DataList list)
```

Appends all the DataFields of the specified DataList to the end of this list. The order of the elements of the sprecified DataList will be preserved.

addAll

```
void addAll (int index, DataList list)
```

Inserts all the DataFields of the specified DataList at the specified position in this list. The order of the elements of the sprecified DataList will be preserved.

clone

```
Object clone ()
```

Returns a deep copy of this DataList.

get

```
DataField get (int index)
```

Returns the element at the specified position in this list.

iterator

```
Iterator iterator ()
```

Returns an iterator over the elements in this list.

remove

```
DataField remove (int index)
```

Removes and returns the DataField at the specified index position.

removeAll

```
void removeAll ()
```

Removes all elements from this list.

size

```
int size ()
```

Returns the number of DataFields of this DataList.

sort

```
void sort (DataList.Comparator listComp)
```

Sorts the list according to the order induced by the specified comparator.

DataList.Comparator

Inerface

DataList.Comparators can be passed to the `sort` method of DataList to allow precise control over the sort order.

Method Summary

int **compare** (DataField f1, DataField f2)

compare

```
int compare (DataField f1, DataField f2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

DataRow

extends DataField
implements Cloneable, Serializable

The class corresponding to the generic Object in ActionScript. When an object is sent to the server, it will be converted to a DataRow instance and each property of the ActionScript object is converted to the corresponding DataField and put into the DataRow instance. DataRow is similar to java.util.Map, but the keys are limited to Strings and the values are limited to DataFields. If you want to map a server-side DataRow to a custom client-side ActionScript class, you can use the `setASClass` method in DataRow. In addition this class must be registered with the same name with the `register` method in the client-side `org.omus.Class` object.

Constructor Summary

DataRow (String name)
DataRow (String name, DataRow dr)

Method Summary

Object **clone** ()
boolean **contains** (String name)
DataField **get** (String name)
String **getASClass** ()
Iterator **iterator** ()
void **put** (DataField df)
void **putAll** (DataRow dr)
DataField **remove** (String name)
void **removeAll** ()
void **setASClass** (String className)
int **size** ()

DataRow

```
DataRow (String name)
```

Creates a new DataRow with the specified name.

DataRow

```
DataRow (String name, DataRow dr)
```

Creates a new DataRow with the specified name and the same DataFields as the specified DataRow.

clone

```
Object clone ()
```

Returns a deep copy of this DataRow.

contains

```
boolean contains (String name)
```

Returns true if this DataRow contains a DataField with the specified name.

get

```
DataField get (String name)
```

Returns the DataField with the specified name or null if no such DataField exists.

getASClass

```
String getASClass ()
```

Returns the registered name of an ActionScript class or null if none has been set.

iterator

```
Iterator iterator ()
```

Returns an iterator over the DataFields in this DataRow.

put

```
void put (DataField df)
```

Adds the specified DataField to this DataRow.

putAll

```
void putAll (DataRow dr)
```

Adds all DataFields of the specified DataRow to this DataRow.

remove

```
DataField remove (String name)
```

Removes and returns the DataField with the specified name. Returns null if no such DataField exists.

removeAll

```
void removeAll ()
```

Removes all DataFields from this DataRow.

setASClass

```
void setASClass (String className)
```

Sets the name of the ActionScript class for this DataRow to the specified string. The objects can only be sent to the client and converted to the corresponding custom ActionScript objects, if a constructor with a corresponding name was registered with the client. See the section about the Class object in the Client API.

size

```
int size ()
```

Returns the number of DataFields of this DataRow.

DataTable

extends DataField
implements Cloneable, Serializable

The class corresponding to `org.omus.Table` in the Client API. It does not have an `updateRow` method like the client-side Table object, because all modified rows will be updated automatically, since all DataField objects keep track of changes.

Constructor Summary

DataTable (String name, TableDefinition td)
DataTable (String name, DataTable dt)

Method Summary

void **addAllRows** (DataTable source)
void **addRow** (DataRow row)
Object **clone** ()
TableDefinition **getDefinition** ()
DataRow **getRow** (int index)
Iterator **iterator** ()
void **removeAllRows** ()
DataRow **removeRow** (int index)
int **size** ()
void **sort** (DataTable.Comparator tableComp)

DataTable

```
DataTable (String name, TableDefinition td)
```

Creates a new DataTable with the specified name and table definition.

DataTable

```
DataTable (String name, DataTable dt)
```

Creates a new DataTable with the specified name and the same rows as the specified table.

addAllRows

```
void addAllRows (DataTable source)
```

Adds all rows of the specified table to this table. This method will only succeed if both tables have matching table definitions.

addRow

```
void addRow (DataRow row)
```

Adds a single row to this table. This method will only succeed if the specified DataRow object matches the table definition. See the entry for the `match` method in TableDefinition for details.

clone

```
Object clone ()
```

Returns a deep clone of the table.

getDefinition

```
TableDefinition getDefinition ()
```

Returns the TableDefinition object used in this table.

getRow

```
DataRow getRow (int index)
```

Returns the row at the specified position.

iterator

```
Iterator iterator ()
```

Returns an iterator over the rows of this table.

removeAllRows

```
void removeAllRows ()
```

Removes all rows from this table.

removeRow

```
DataRow removeRow (int index)
```

Removes and returns the row at the specified index.

size

```
int size ()
```

Returns the number of rows of this table.

sort

```
void sort (DataTable.Comparator tableComp)
```

Sorts the table according to the order induced by the specified comparator.

DataTable.Comparator

Interface

DataTable.Comparators can be passed to the sort method of DataTable to allow precise control over the sort order.

Method Summary

int **compare** (DataRow r1, DataRow r2)

compare

```
int compare (DataRow r1, DataRow r2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

DateField

extends DataField
implements Cloneable, Serializable

The class corresponding to Date objects in ActionScript.

Constructor Summary

DateField (String name, Date value)

Method Summary

Date **getDate** ()
void **setDate** (Date newValue)

DateField

```
DateField (String name, Date value)
```

Creates a new DateField with the specified name and value.

getDate

```
Date getDate ()
```

Returns the value of this DateField.

setDate

```
void setDate (Date newValue)
```

Sets the value of this DateField to the specified Date.

DoubleField

extends DataField
implements Cloneable, Serializable

The class corresponding to the number datatype in ActionScript.

Constructor Summary

DoubleField (String name, double value)

Method Summary

double **getDouble** ()
void **setDouble** (double newValue)

DoubleField

```
DoubleField (String name, double value)
```

Creates a new DoubleField with the specified name and value.

getDouble

```
double getDouble ()
```

Returns the value of this DoubleField as a double.

setDouble

```
void setDouble (double newValue)
```

Sets the value of this DoubleField to the specified double.

FieldListener

Interface

A listener that gets notified when the value of a DataField changes. Can be registered with any DataField through its `addListener` method.

Method Summary

void **fieldChange** (DataField field)

fieldChange

```
void fieldChange (DataField field)
```

Invoked when the value of the specified DataField was modified.

FieldVisitor

Interface

A visitor that can be used to extend the functionality of all DataFields without modifying their API. Can be passed to every DataField through its `accept` method.

Method Summary

void **visit** (BooleanField bf)
void **visit** (CounterField cf)
void **visit** (DataList dl)
void **visit** (DataRow dr)
void **visit** (DataTable dt)
void **visit** (DateField df)
void **visit** (DoubleField df)
void **visit** (IntField inf)
void **visit** (LongField lf)
void **visit** (StringField sf)

Method details are omitted; each method visits the specified field.

IntField

extends DataField
implements Cloneable, Serializable

A class for properties or table columns configured as an int, corresponding to the number datatype in ActionScript. This type of DataField will only be created if a property or column in a table is explicitly configured as an int. All other ActionScript numbers that are converted automatically will be converted to DoubleFields.

Constructor Summary

IntField (String name, int value)

Method Summary

int **getInt** ()
void **setInt** (int newValue)

IntField

```
IntField (String name, int value)
```

Creates a new IntField with the specified name and value.

getInt

```
int getInt ()
```

Returns the value of this IntField as an int.

setInt

```
void setInt (int newValue)
```

Sets the value of this IntField to the specified int.

LongField

extends DataField
implements Cloneable, Serializable

A class for properties or table columns configured as a long, corresponding to the number datatype in ActionScript. This type of DataField will only be created if a property or column in a table is explicitly configured as a long. All other ActionScript numbers that are converted automatically will be converted to DoubleFields.

Constructor Summary

LongField (String name, long value)

Method Summary

long **getLong** ()
void **setLong** (long newValue)

LongField

```
LongField (String name, long value)
```

Creates a new LongField with the specified name and value.

getLong

```
long getLong ()
```

Returns the value of this LongField as a long.

setLong

```
void setLong (long newValue)
```

Sets the value of this LongField to the specified long.

Message

implements Cloneable, Serializable

A message that can be sent to Flash clients or to other users and groups on the server. It corresponds to the client side Message object. The attachment is a DataRow which will be converted to a generic ActionScript object in the client and thus can carry any of the ActionScript datatypes that correspond to one of the server side DataField objects.

Constructor Summary

Message (String subject)
Message (String subject, User sender)

Method Summary

Object **clone** ()
DataRow **getAttachment** ()
String **getSender** ()
String **getSubject** ()
Command **getUndeliveredCommand** ()
void **setUndeliveredCommand** (Command com)

Message

```
Message (String subject)
```

Creates a new message with the specified subject.

Message

```
Message (String subject, User sender)
```

Creates a new message with the specified subject and the specified user set as the sender.

clone

```
Object clone ()
```

Returns a clone of this message containing a deep copy of its attachment.

getAttachment

```
DataRow getAttachment ()
```

Returns the attachment of this message. Initially it is an empty DataRow, but you can put any DataField into it and all the values will be converted to their corresponding ActionScript datatypes, if you send the message to a client.

getSender

```
String getSender ()
```

Returns the name of the sender.

getSubject

```
String getSubject ()
```

Returns the subject of this message.

getUndeliveredCommand

```
Command getUndeliveredCommand ()
```

Returns the command that gets executed if the message can not be delivered to the recipient or null if none has been specified yet.

setUndeliveredCommand

```
void setUndeliveredCommand (Command com)
```

Sets the command that gets executed if the message can not be delivered to the recipient. This is intended for messages that have a target on the server side. Note that the Command must be serializable if the Oregano Server runs on more than one machine.

StringField

extends DataField
implements Cloneable, Serializable

The class corresponding to String objects in ActionScript.

Constructor Summary

StringField (String name, String value)

Method Summary

String **getString** ()
void **setString** (String newValue)

StringField

```
StringField (String name, String value)
```

Creates a new StringField with the specified name and value.

getString

```
String getString ()
```

Returns the value of this StringField.

setString

```
void setString (String newValue)
```

Sets the value of this StringField to the specified string.

TableDefinition

implements Cloneable, Serializable

TableDefinition objects are used to define the names and datatypes of the columns of a DataTable object.

Constructor Summary

TableDefinition ()

Method Summary

void **addColumn** (String name, Class type)
Object **clone** ()
boolean **equals** (Object o)
int **getColumnCount** ()
String **getColumnName** (int index)
String[] **getColumnNames** ()
Class **getColumnType** (int index)
boolean **matches** (DataRow row)

TableDefinition

```
TableDefinition ()
```

Creates a new table definition.

addColumn

```
void addColumn (String name, Class type)
```

Adds a new column to this table definition. This method will fail if this TableDefinition object was already used to construct a new DataTable object, since it is unmodifiable after that. The second argument must be one of the 10 subclasses of DataField.

clone

```
Object clone ()
```

Returns a clone of this table definition.

equals

```
boolean equals (Object o)
```

Returns true if the specified object is a table definition with the same sequence of columns.

getColumnCount

```
int getColumnCount ()
```

Returns the number of columns in this table definition.

getColumnName

```
String getColumnName (int index)
```

Returns the name of the column at the specified index position.

getColumnNames

```
String[] getColumnNames ()
```

Returns an array of all column names in the order they were added to this TableDefinition.

getColumnType

```
Class getColumnType (int index)
```

Returns the type of the column at the specified index position.

matches

```
boolean matches (DataRow row)
```

Returns true if the specified DataRow matches this table definition. The number of DataFields contained in the specified DataRow must match the number of columns in this definition object and each DataField of the specified DataRow must have a matching name and datatype in this definition object.

Package org.omus.db

Contains classes for database connectivity. Can be used for reading and writing DataField objects from and to the database.

DbException

implements Serializable

Thrown when an error occurs during execution of a transaction. You will rarely create a DbException object yourself, but many methods in the Oregano Server API throw this Exception, which you have to catch in your extensions. You can use the `getMessage` method inherited from Throwable to obtain the client-side error code of this Exception. See the section on error codes at the end of the Client API chapter for a list of possible error codes.

Constructor Summary

DbException (String errorCode)

DbException

```
DbException (String errorCode)
```

Creates a new DbException with the specified error code.

DbReader

extends DbTransactionPart
implements Serializable

Represents a `SELECT` statement to be executed on the server.

Constructor Summary

DbReader (String configID)
DbReader (String configID, String resultID)

DbReader

```
DbReader (String configID)
```

Creates a new DbReader object. The specified configID must match a configID attribute in a `<dbReader>` node in dbCustom.xml.

DbReader

```
DbReader (String configID, String resultID)
```

Creates a new DbReader object. The specified configID must match a configID attribute in a <dbReader> node in dbCustom.xml. The specified resultID can be used to read values from the DbResult object.

DbResult

implements Serializable

Represents the result of a transaction.

Method Summary

int **getAffectedRows** (String resultID)
Message **getMessage** (String subject)
DataTable **getTable** (String resultID)

getAffectedRows

```
int getAffectedRows (String resultID)
```

Returns the number of affected rows of the statement with the specified resultID. The resultID must match the ID that you passed as the second argument to the constructor of the corresponding DbWriter object.

getMessage

```
Message getMessage (String subject)
```

Returns a message with the specified subject and the content of this DbResult object as an attachment. A convenient short cut if you want to send the values that you read from the database to a client.

getTable

```
DataTable getTable (String resultID)
```

Returns a table object read from the database. The resultID must match the ID that you passed as the second argument to the constructor of the corresponding DbReader object.

DbTransaction

implements Serializable

Represents a transaction; contains one or more DbWriter or DbReader objects to execute custom SQL statements. The statements must be configured in the dbCustom.xml file on the server. This approach may seem a bit cumbersome at first sight. But it is very versatile once you get used to it.

The following example illustrates how you can use the DbTransaction object to read a user ID from one table and use that ID to change a value in another table. By the way: It is the same example used to illustrate the use of the client-side DbTransaction object. Most of the classes in `org.omus.db` have their twins in the Client API.

The corresponding entries in dbCustom.xml might look as follows (see the Configuration chapter for more details):

```
<dbReader configID="readUserID" rollbackOnError="true" minRows="1">
    <in name="username" />
    <out name="userID" />
    <statement>
        SELECT userID FROM reginfo WHERE username = ?
    </statement>
</dbReader>

<dbWriter configID="updateAddress" rollbackOnError="true">
    <in name="street" />
    <in name="city" />
    <in name="phone" />
    <in name="userID" />
    <statement>
        UPDATE address SET street = ?, city = ?, phone = ?
        WHERE userID = ?
    </statement>
</dbWriter>
```

You can write Java code to execute those statements as follows:

```
DbTransaction ta = new org.omus.DbTransaction();
ta.add(new org.omus.DbReader("readUserID"));
ta.add(new org.omus.DbWriter("updateAddress","address"));
ta.setParam(new StringField("username","Thomas"));
ta.setParam(new StringField("street","217 Mulholland Drive"));
ta.setParam(new StringField("city","Los Angeles"));
ta.setParam(new StringField("phone","27 27 55 55"));
DbResult res = null;
try {
    res = ta.execute();
    int afRows = res.getAffectedRows("address");
    // should be 1
} catch (DbException dbe) {
    // handle Exception
}
```

Please note that the second statement has 4 <in> nodes. Three of them (street, city and phone) will be set by your Java code. The last one (userID) will be set automatically because it is configured as an <out>-value in the first statement which will be available to all subsequent statements. The attribute minRows is set to "1" in the first statement so that execution of the transaction will be aborted if the result is empty, because the second statement would fail anyway, if no user with the specified name exists.

Constructor Summary

DbTransaction ()

Method Summary

void **add** (DbTransactionPart tap)
DbResult **execute** () throws DbException
void **setAllParams** (DataRow dr)
void **setParam** (DataField df)
int **size** ()

DbTransaction

```
DbTransaction ()
```

Creates a new DbTransaction object.

add

```
void add (DbTransactionPart tap)
```

Adds a DbWriter or DbReader object to this transaction.

execute

```
DbResult execute () throws DbException
```

Executes this transaction and returns the result.

setAllParams

```
void setAllParams (DataRow dr)
```

Adds all DataFields contained in the specified DataRow as a parameter to this DbTransaction. The name of each DataField must have a matching name attribute in one of the <in> nodes you specified in the dbCustom.xml file. If you set parameters in the DbTransaction object, they are available to all the DbWriter or DbReader parts you add to this transaction. If you want to set parameters that can only be read by one particular DbWriter or DbReader object, you must use the `setParam` method of those objects.

setParam

```
void setParam (DataField df)
```

Adds the specified DataField as a parameter to this DbTransaction. The name of the DataField must match the name attribute of one of the <in> nodes you specified in the dbCustom.xml file. If you set a parameter in the DbTransaction object it is available to all the DbWriter or DbReader parts you add to this transaction. If you want to set parameters that can only be read by one particular DbWriter or DbReader object, you must use the `setParam` method of those objects.

size

```
int size ()
```

Returns the number of DbWriter and DbReader objects that have been added to this transaction.

DbTransactionPart

implements Serializable

Abstract superclass of DbWriter and DbReader.

Method Summary

void **setAllParams** (DataRow dr)
void **setParam** (DataField df)

setAllParams

```
void setAllParams (DataRow dr)
```

Adds all DataFields contained in the specified DataRow as a parameter to this DbTransactionPart. The name of each DataField must have a matching name attribute in one of the <in> nodes you specified in the dbCustom.xml file.

setParam

```
void setParam (DataField df)
```

Adds the specified DataField as a parameter to this DbTransactionPart. The name of the DataField must match the name attribute of one of the <in> nodes you specified in the dbCustom.xml file.

DbWriter

extends DbTransactionPart
implements Serializable

Represents a `DELETE`, `UPDATE` or `INSERT` statement to be executed on the server.

Constructor Summary

DbWriter (String configID)
DbWriter (String configID, String resultID)

DbWriter

```
DbWriter (String configID)
```

Creates a new DbWriter object. The specified configID must match a configID attribute in a `<dbWriter>` node in dbCustom.xml.

DbWriter

```
DbWriter (String configID, String resultID)
```

Creates a new DbWriter object. The specified configID must match a configID attribute in a `<dbWriter>` node in dbCustom.xml. The specified resultID can be used to check the number of affected rows in the DbResult object.