In a nutshell

Terry Dye, Andreas Mecky

Inhaltsverzeichnis

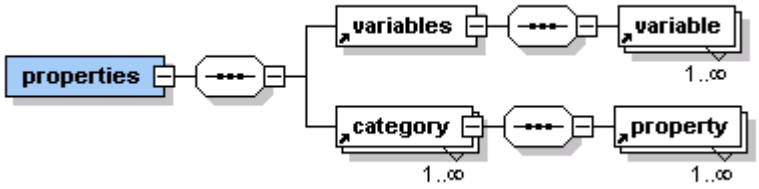
1 The basics

This guide should help you get started with jConfig. We will go through some steps that will demonstrate most, if not all, of the features.

1.1 First steps

First of all we will define a very simple configuration file called config.xml:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<properties>
  <variables>
    <variable name="my.path" value="/home/foo/data"/>
  </variables>
  <category name="general">
    <property name="upload_dir" value="${my.path}/data"/>
    <property name="NewsCounter" value="10"/>
    <property name="showNews" value="true"/>
    <property name="MyProp" value="Hello world"/>
  </category>
  <category name="JDBC">
    <property name="URL" value="jdbc:mysql://localhost/iportal"/>
    <property name="DRIVER" value="org.gjt.mm.mysql.Driver"/>
    <property name="PWD" value="pwd"/>
    <property name="USER" value="user"/>
  </category>
</properties>
```



Save this file and make sure that you have it in your classpath.

Now we will write the "Hello world" application:

```
import org.jconfig.*;

public class ConfigDemo {

       private static final Configuration configuration =
ConfigurationManager.getConfiguration();

    public static void main(String[] args) {
        String myProp = configuration.getProperty("MyProp");
        System.out.println("MyProp:"+myProp);
    }
}
```

In order to run all the demos make sure you have the config.xml from above, the jconfig.jar, crimson.jar and the jaxp.jar in your classpath. We have provided a shell script to compile the file:

```
./compile.sh ConfigDemo
```

There is a little shell script that sets the classpath correct and you can use it to run the demo:

```
./run.sh ConfigDemo
```

This little demo will write out "Hello world". Actually not so impressive, but shows how things can work. What happens behind the scenes is that the ConfigurationManager tries to find the file config.xml in the classpath and if it is found it will read in your configuration automatically.

You do not have to load the configuration. When you call the

```
getConfiguration()
```

method then the ConfigurationManager returns the default configuration which in our case is the configuration inside our config.xml. When you use the

```
getProperty(String name)
```

or

```
getProperty(String name,String defaultValue)
```

methods then jConfig will return the property value from the default category. In order to get the property value from a specific category we use the

```
getProperty(String name,String defaultValue,String category)
```

Now we will update our little example from above and use these three methods:

```
public static void main(String[] arg) {
    String myProp = configuration.getProperty("MyProp");
    System.out.println("MyProp:"+myProp);
    String jdbcUser = configuration.getProperty("USER",null,"JDBC");
    System.out.println("jdbcUser:"+jdbcUser);
    int newsCounter = configuration.getIntProperty("NewsCounter",-1);
    if ( newsCounter == 10 ) {
        System.out.println("We have found the correct value");
    }
    boolean showNews = configuration.getBooleanProperty("showNews",false);
    if ( showNews ) {
        System.out.println("We have to show the news");
    }
}
```

Run this demo with the same command as above.

1.2 Defining a configuration name

So far we have used a file called config.xml for our configuration and we have called

```
ConfigurationManager.getConfiguration()
```

to get our configuration. The ConfigurationManager can handle many configurations which must have an unique name. In order to get this configuration you have to call:

```
ConfigurationManager.getConfiguration("myapp");
```

The ConfigurationManager will now try to find a file called myapp_config.xml automatically. This is the naming convention that is used. Simply add "_config.xml" to your configuration name and you have the file name that the ConfigurationManager will search for. If no file is found then it will try to read a file called config.xml. When there is no file then it will create a new configuration. This configuration is generated with the given name.
Note: In order to see whether this is a new configuration or it was loaded you can call (since v2.5):

```
Configuration config = ConfigurationManager.getConfiguration("myapp");
if ( config.isNew() ) {
  System.out.println("This config was NOT loaded. It was created");
}
```

The **isNew** flag is reset automatically when a configuration could be loaded.

1.3 Support for primitives

Since not all of your configuration data are Strings the configuration supports convenience methods to access java primitives and String arrays. Here is an XML example that we will use here:

```
<category name="example">
  <property name="intValue" value="1000"/>
  <property name="list" value="Mon,Tue,Wed,Thu,Fri"/>
  <property name="money" value="100.40"/>
</category>
```

Getting the int value means:

```
int myValue = config.getPropety("intValue",-1,"example");
```

You have to define a default value every time when you use one of the convenience methods. Since the configuration does not know which default value to return if the property is not found you have to specify it. In order to get the list you can do the following:

```
String[] myArray = config.getProperty("list",new String[]{"Huhu"},"example");
System.out.println("day 3 is:"+myArray[2]);
```

The configuration support all java primitives. If we have forgotten one then let us know.

1.4 Inheritance

Our next demo will show the inheritance that jConfig supports. If you try to get the property from a category that does not contain this property then the Configuration will look for the property in the default category.

```
import org.jconfig.*;

public class InheritanceDemo {

    private static final Configuration configuration =
        ConfigurationManager.getConfiguration();

    public static void main(String[] args) {
        String myProp = configuration.getProperty("MyProp",null,"JDBC");
        System.out.println("MyProp:"+myProp);
    }
}
```

1.5 Variables

We have defined a variable in our configuration and now it is time to explain it. In order to ease the usage of often repeated phrases inside property values, jConfig offers the possibility to define variables. These variables are replaced at runtime with the defined values when you call getProperty(....). Here is a short example:

```
<properties>
  <variables>
    <variable name="base_path" value="/home/foo/application"/>
  </variables>
  <category name="general">
    <property name="upload_dir" value="${base_path}/data"/>
  </category>
</properties>
```

The source code for this example is the same as we have used so far:

```
import org.jconfig.*;
```

```
public class VariableDemo {

    private static final Configuration configuration =
        ConfigurationManager.getConfiguration();

    public static void main(String[] args) {
        String uploadDir = configuration.getProperty("upload_dir");
        System.out.println("upload_dir:"+uploadDir);
    }
}
```

This will write out:

```
uploadDir:/home/foo/application/data
```

Since version 2.7 you can use variables inside variable definitions like this:

```
<variables>
  <variable name="base_path" value="/home/foo/application"/>
  <variable name="data_path" value="${base_path}/data"/>
  <variable name="context_path" value="${base_path}/context"/>
</variables>
```

There is more about variables written in the chapter AdvancedVariables

1.6 PropertyListener

One of the new features is the PropertyListener that is raised every time the value of one of the properties is changed or a new property is added to a category. In order to receive this event you have to implement the PropertyListener interface:

```
import org.jconfig.*;
import org.jconfig.event.*;

public class PropertyListenerDemo implements PropertyListener {

    private static final Configuration configuration =
ConfigurationManager.getConfiguration();

    public static void main(String[] args) {
        configuration.addPropertyListener(new PropertyListenerDemo());
        configuration.setProperty("Does not matter","hello world");
    }

    public void propertyChanged(PropertyListenerEvent e) {
        System.out.println("property '"+e.getPropertyName()+"' in category
'"+e.getCategoryName()+"' has changed");
    }
}
```

1.7 Loading and saving

The last step is loading and saving a configuration file. There are some handlers that you can use for loading and saving your configuration files. You can find more about handlers in the chapter „Handlers“

```
import org.jconfig.handler.*;
import org.jconfig.*;
import java.io.File;

public class LoadSaveDemo {

    private static final ConfigurationManager cm =
        ConfigurationManager.getInstance();

    public static void main(String[] args) {
```

```
        File file = new File("test.xml");
        XMLFileHandler handler = new XMLFileHandler();
        handler.setFile(file);
        try {
            System.out.println("trying to load file");
            cm.load(handler,"myConfig");
            System.out.println("file successfully processed");
            Configuration config = cm.getConfiguration("myConfig");
            config.setProperty("TEST","Added property");
            System.out.println("trying to save file");
            cm.save(handler,config);
            System.out.println("file successfully saved");
            System.out.println("TEST:"+config.getProperty("TEST"));
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Note: As of version v2.3 we have made saving easier. All you need is the call the:

```
cm.save("myConfig");
```

The ConfigurationManager keeps the associated handler in memory and will call the save method of the handler. If you want to use a different handler then you need to call the

```
cm.save(handler,configname);
```

For example if you have used the URLHandler to load a configuration file from an URL and want to save it to a local file to have a backup.

2 The configuration server

Managing configurations can get difficult. When you run your application in different environments for example. Or if you have a lot of clients that all share the same configuration. When you include the configuration in the client installation then every update is difficult.
The configuration server can help a lot in these cases.

2.1 How does the server works

The configuration server is mainly a web server. It uses the HTTP protocol to communicate between the client and server. The server has a strategy based on the incoming ip-address and the configuration name to find the configuration. One important thing is how the server will actually search for a configuration when a request comes in. First of all the server will try to find a configuration file with the name based on the ip-address of the client and the requested configuration name. So it will look for (example):

```
127.0.0.1_myconfig.xml
```

If not found it will cut off the last part of the ip-address and try again. It will repeat this until there only the name left. The reason for this lookup mechanism is that you can move your application from one server to another server and the config server will give you the correct configuration. This makes moving an application from a test to a development server much easier. The application does not change and does not have to take care of different configurations.

2.2 Installing the server

First create a directory where the server should run in. Then you can create a subdirectory where the configuration files will be stored. This is optional. Copy the jconfig.jar file to this directory. Create the jconfig.properties file. Now you can run the server:

```
java -classpath .;jconfig.jar org.jconfig.server.ConfigurationServer -port x -docroot dir
```

Please set the x and the dir to the desired values. If the port is not set it will be set to 8234 and the docroot will be set to the current system tmp-directory if not defined.

2.3 Running the server

In order to run the server it needs two parameters. The port and where the base directory is where all configurations are stored. You can define the parameters inside a file "jconfig.properties" and put this file in the classpath:

```
# the port that the server will listen on
server.port=8765
# the directory where the server will search the configuration files
server.docroot=c:\configserver\files
```

2.4 The ConfigServerHandler

In order to get a configuration from the server the client has to use the ConfigServerHandler. Here is an example:

```
import org.jconfig.*;
import org.jconfig.handler.*;

public class MyConfigLoader {

    public static void load() {
```

```
      ConfigServerHandler handler = new ConfigServerHandler();
      try {
        urlHandler.setURL("jconfig://myconfigserver:8765/serverconfig");
        ConfigurationManager cm = ConfigurationManager.getInstance();
        // we use the ConfigurationManager so it can store the config
        cm.load(urlHandler.load(),"MyConfig");
      }
      catch (ConfigurationManagerException cme) {
        // do something here since we did not get a configuration
      }
    }
}
```

In this example we have used the ConfigServerHandler to load the configuration. If we use the load method in the ConfigurationManager then the configuration will be stored and all other classes can access the configuration simply like this:

```
import org.jconfig.*;

public class MyApp {

    private static final Configuration config = ConfigurationManager.getConfiguration
("MyConfig");

    public static void main(String[] args) {
      System.out.println("config:"+config.toString());
    }
}
```

We can also implement a fall back strategy with the first example. The way would be to load the configuration and save it to a file. If the next time we cannot load the configuration we have at least the backup of the last time. Here is an example:

```
import org.jconfig.*;
import org.jconfig.handler.*;
import java.io.File;

public class MyConfigLoader {

    public static void load() {
      ConfigServerHandler handler = new ConfigServerHandler();
      try {
        urlHandler.setURL("jconfig://myconfigserver:8765/serverconfig");
        ConfigurationManager cm = ConfigurationManager.getInstance();
        // we use the ConfigurationManager so it can store the config
        cm.load(urlHandler.load(),"MyConfig");
        File file = new File(System.getProperty("java.io.tmpdir")+"config_backup.xml");
        XMLFileHandler fileHandler = new XMLFileHandler();
        fileHandler.setFile(file);
        cm.store(fileHandler,"MyConfig");
      }
      catch (ConfigurationManagerException cme) {
        // we will use our backup file
        try {
          File file = new File(System.getProperty("java.io.tmpdir")+"config_backup.xml");
          XMLFileHandler fileHandler = new XMLFileHandler();
          fileHandler.setFile(file);
          cm.load(fileHandler.load(),"MyConfig");
        }
        catch (ConfigurationManagerException cme) {
          // ok, now we are doomed
        }
      }
    }
}
```

2.5 The configuration server as MBean

Running the configuration server as a standalone process is not the only option. In order to easily integrate it in your J2EE environment there is also a MBean available. We will ow

describe how to install the server inside the JBoss application server. If you know how to setup the server in a different server then please send us an email.

Here is a sample configuration which works in JBoss-3.x:

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <classpath codebase="lib" archives="jconfig.jar"/>
  <mbean code="org.jconfig.server.ConfigurationServer"
         name="user:configuration=ConfigurationServer">
    <attribute name="Port">8765</attribute>
    <attribute name="Daemon">true</attribute>
    <attribute name="DocumentRoot">/home/mecky/jboss-
3.0.3/server/default/conf/</attribute>
    <attribute name="Started">true</attribute>
  </mbean>
</server>
```

Please put the jconfig.jar in the lib-directory of the server. In this example it is:
`/home/mecky/jboss-3.0.3/server/default/lib`

It is important to set the „daemon" parameter to „true" since this way the configuration server will be shutdown with JBoss since it runs as a child daemon of the server. The „started" attribute is a fake in order to implement a startup method. This MBean is a generic MBean and therefore does not include any of the JBoss-MBean interfaces.

3 More about Variables

The chapter about the basics has provided an overview about how to use variables inside the configuration. Here are two more things that eases the use of variables and help you writing a platform independent configuration.

3.1 System properties

Since the latest release v2.3 the VariableManager can translate java system properties. These system properties are set by the JVM and are mostly platform dependent. The temp-directory for example differs for every operating system. In order to build a platform independent configuration you can use these system properties. You need to put the system**:** keyword in front of the property name:

```
<property name="tmpdir" value="${system:java.io.tmpdir}"/>
```

After the keyword you have to put the name of the system property. This example will set the tmpdir to the OS specific temp-directory. You can use any system property.

3.2 Environment variables

Another way for writing platform independent applications is the use of environment variables. This is an usual at least under Linux/Unix. You can set a home variable to your application. When you want to use an environment variable you have to put the **env:** keyword in front of the name. Here is a short example:

```
<property name="app_home" value="${env:MY_APP_HOME}"/>
```

This **MY_APP_HOME** can be either "/usr/local/myapp" under Linux or "c:\myapp" under Windows. All you have to do is define such an environment variable and you can move your applications between different OS.

4 Including properties

Sometimes configurations only differ in a few settings. For example when several developers work on the same project or everyone has it own settings for a specific path. Instead of defining an individual configuration you can now extract these properties and put it into a separate properties file. The configuration will be the same for all but the properties file is different. Another example would be when you deploy your application in a test or production environment.

4.1 How does it work

You can define the name of the properties file and this file will be read by the parser and these properties will be set as immutable variables. The variables cannot be overwritten and will not be saved when you save a configuration. Even if you would use the VariableManager to set such a variable it will not be saved. Otherwise this would break the entire functionality.

4.2 Example

This is a short example demonstrating how to use such a properties file.
First we define two properties in a file called base.properties

```
path=/tmp/mypath
serverroot=/usr/local/mydocs
```

Make sure that this file is in your classpath. Now we can define the XML:

```
<?xml version="1.0"?>
<properties>
  <include properties="base.properties"/>
  <category name="general">
    <property name="base.path" value="${path}"/>
    <property name="server.path" value="${serverroot}"/>
  </category>
</properties>
```

The interesting part is marked as bold in the example above. That is all you need to do. Now the configuration will be the same and the few differences are stored in a properties file. This approach of course will only make sense when there are only a few differences.

5 Inheritance for configurations

As of version v2.8 you can now extends configurations. There is only single inheritance supported.

5.1 When to use it

There is often the case that a configuration differs in a few properties when used in different environments or shared between different developers. One way of handling these differences is to put the differences into a properties file and include it in the configuration. You can read more about it in the previous chapter. Another way is to build a common configuration as a base and then extend this configuration and overwrite the special properties. Using this approach it is also possible to build a common configuration that might be used in different projects where only a basic set of properties are the same.

5.2 How to use it

In order to define that a configuration extends another one you have to define it in the properties tag like this:

<?xml version="1.0"?>

```
<properties extends="base">
....
</properties>
```

This means that this configuration extends the configuration called **base**. Following the naming convention of jConfig this means that this configuration extends a configuration that it read from a file called *base_config.xml*.

5.3 How does it work

When you try to get a property from a configuration the Configuration will first of all try to find it in the specific category. If the propery is not found then it will look inside the main category (called "default").

Still not found then it will call the base configuration and try to get the property from it. If still not found it will return the default value.

5.4 Circular dependencies

Using inheritance can lead to an circular dependeny. For example if config A would extend config B and config B would extend config A. The AbstractConfigParser takes care of this and will check if there is such a dependency before the base configuration name is set. If there is such a case the inheritance will not take effect. You will get an error reporting this when this happens.

5.5 One short example

First we create a file called *base_config.xml*:

```
<?xml version="1.0"?>
<properties>
  <category name="default">
    <server>10.1.100.18</server>
    <port>8080</port>
    <user_required>true</user_required>
    <base_path>/usr/local/data</base_path>
  </category>
</properties>
```

This will be the base for our confguration. Now let us write a file called **app_config.xml**:

```
<?xml version="1.0"?>
```

```
<properties extends="base">
  <category name="default">
    <server>10.1.100.22</server>
    <base_path>/home/amecky/data</base_path>
  </category>
</properties>
```

When you now get the configuration **app** like this:

```
Configuration config = ConfigurationManager.getConfiguration("app");
```

and you try to get the property **server** it will return **10.1.100.22** since you have overwritten this property. If you try to get **port** it will return **8080** since it is defined in the base configuration.

5.6 Limitations

Currently the variables are not extended. This means that you cannot access the variables from the base configuration.

6 The handlers

A handler is sued to load a configuration from a file or something else. There are a set of different handlers included. The ConfigurationManager uses the InputStreamHandler as default handler. When the handler has loaded the configuration it will call the specified parser to build the configuration.

6.1 The handlers described

In the following we will describe the handlers that are supported by jConfig.

6.2 InputStreamHandler

This handlers tries to find the desired file in the classpath. It uses the ResourceLocator in order to search for it. It is the default handler. This is a source example about how to use it:

```
public void testLoad() {
   InputStreamHandler cHandler = new InputStreamHandler();
   cHandler.setFileName("test.xml");
   try {
      Configuration config = cHandler.load("test");
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}
```

6.3 PropertiesFileHandler

This handler will read in a properties file. It uses also the ResourceLocator to find the properties file. The handler can be used to convert old properties files into a configuration and then use the XMLFileHandler to store them as an XML file.
TODO: include example here

6.4 ScriptHandler

This handler is more like an experiment. We are currently thinking about if it might make sense introducing a script language for configurations.

6.5 URLHandler

If you want to load the configuration from a web server then you can use this one. Here is a short example:

```
import org.jconfig.*;
import org.jconfig.handler.*;

public class URLDemo {

  public static void main(String[] args) {
     try {
       URLHandler handler = new URLHandler();
       handler.setURL("http://localhost:8181/test.xml");
       ConfigurationManager cm = ConfigurationManager.getInstance();
       cm.load(handler,"test");
     }
     catch (Exception e) {
       e.printStackTrace();
     }
  }
}
```

Note: If you use the ConfigurationManager to load the configuration then it will keep it in and you can now use:

```
Configuration config = ConfigurationManager.getConfiguration("test");
```

to access it.

6.6 XMLFileHandler

The XMLFileHandler will load a XML file and you have to define the full path and filename. Here is an exam:

```
public void testLoadAndSave() {
   File file = new File("/home/app/my_config.xml");
   try {
       XMLFileHandler handler = new XMLFileHandler();
       handler.setFile(file);
       Configuration config = handler.load("test");
   }
   catch (Exception e) {
       e.printStackTrace();
   }
}
```

This will load the file my_config.xml from the defined directory and parse the content building a new configuration called „test".

6.7 ConfigServerHandler

This one is used to communicate with the ConfigurationServer. This is an example about how to use it:

```
java.net.URL jcfURL = null;
try {
    ConfigServerHandler urlHandler = new ConfigServerHandler();
     urlHandler.setURL("jconfig://localhost:8765/config");
    Configuration config = urlHandler.load();
}
catch ( Exception e ) {
    e.printStackTrace();
 }
```

6.8 DatabaseHandler

This implementation provides the basic requirements for a JDBC Handler. The database structure is quite simple. It is a based on a Configuration table:
`T_CONFIGURATION`
a Category table
`T_CATEGORY`
a Property table
`T_PROPERTY`
and a Variable table
`T_VARIABLE`

A Configuration can have 1:many Categories and 1:many Variables and a category can have 1:many Properties.

Implementation does require that the database support some sort of auto-increment for the object identifier (oid), if you want to store the Configurations.

One problem with the implementation is the ability to store the variable information within the properties. This is an inherent problem with all of the ConfigurationHandlers.

Following properties need to set inside of the jconfig.properties file:

```
org.jconfig.jdbc.driver
org.jconfig.jdbc.user
```

```
org.jconfig.jdbc.pwd
org.jconfig.jdbc.url
```

The SQL create statements for various databases are include in the distribution. Here is an example for MySQL:

```
/*==============================================================*/
/* Database name:  PHYSICALDATAMODEL_2                          */
/* DBMS name:      MySQL 3.23                                   */
/* Created on:     12.04.2004 13:40:38                          */
/*==============================================================*/


drop table if exists T_CATEGORY;
drop table if exists T_CONFIGURATION;
drop table if exists T_PROPERTY;
drop table if exists T_VARIABLE;

create table if not exists T_CATEGORY
(
   OID                            int                            not null
AUTO_INCREMENT,
   NAME                           varchar(255),
   CONFIGURATION_OID              int,
   primary key (OID)
);

create table if not exists T_CONFIGURATION
(
   OID                            int                            not null AUTO_INCREMENT,
   NAME                           varchar(255)                   not null,
   primary key (OID),
   unique (NAME)
);

create table if not exists T_PROPERTY
(
   OID                            int                            not null AUTO_INCREMENT,
   NAME                           varchar(255),
   VALUE                          varchar(255),
   CATEGORY_OID                   int                            not null,
   primary key (OID)
);

create table if not exists T_VARIABLE
(
   OID                            int                            not null AUTO_INCREMENT,
   NAME                           varchar(255),
   VALUE                          varchar(255),
   CONFIGURATION_OID              int,
   primary key (OID)
);
```

There are several create scripts included in the sql subdirectory of the source dustribution.

6.9 More about loading and saving

The examples in the chapters above have demonstrated how to use each handler. In general the best way to load a configuration using one of the handlers is to use the ConfigurationManager to handle this. Using this approach will make sure that the ConfigurationManager can cache the configuration and you can easily save the configuration as well. Here is a short example how to use the XMLFileHandler and the ConfigurationManager:

```
import org.jconfig.*;
import org.jconfig.handler.*;
import java.io.File;

public class URLDemo {
```

```
  public static void main(String[] args) {
     try {
       File file = new File("/home/app/my_config.xml");
       XMLFileHandler handler = new XMLFileHandler();
       handler.setFile(file);
       ConfigurationManager cm = ConfigurationManager.getInstance();
       cm.load(handler,"test");
     }
     catch (Exception e) {
       e.printStackTrace();
     }
  }
}
```

If you use the ConfigurationManager to load the configuration then it will keep it in and you can now use:

```
Configuration config = ConfigurationManager.getConfiguration("test");
```

to access it. Since the ConfigurationManager will cache the configuration and remember which handlers was used to load this configuration it is easy to save the configuration:

```
import org.jconfig.*;
import org.jconfig.handler.*;

public class URLDemo {

  public static void main(String[] args) {
     try {
       ConfigurationManager cm = ConfigurationManager.getInstance();
       cm.save("test");
     }
     catch (Exception e) {
       e.printStackTrace();
     }
  }
}
```

6.10 Writing your own handler

TODO: explain it

7 Different XML structures and different parsers

jConfig supports a simple XML structure which consists of a two level hierarchy. There has been a few requests that this structure is not flexible enough. The latest version now supports different XML structure through the pluggable parsers. Beside the parser there is now the possibility to include different configurations. Currently jConfig supports three different kind of parsers. Every parser can read a different kind of XML structure.

7.1 DefaultConfigParser

This is the default parser. It supports a two level structure with categories and properties inside the categories.
Example:

```
<properties>
  <variables>
    <variable name="base_dir" value="${system:java.io.tmpdir}"/>
  </variables>
  <category name="general">
    <property name="hello" value="world"/>
  </category>
  <category name="server">
    <property name="port" value="8080"/>
    <property name="docroot" value="${base_dir}/docs"/>
  </category>
</properties>
```

There is an example included in the demo subdirectory showing the basic usage.

7.2 CDataConfigParser

The default parser has a few disadvantages. First of all it does not support escape characters as property values. Also the definition of a property might be a little to complicated.

This parser is based on the structure of the default parser but it takes the tag name as property name and the CData content as value.

Example:

```
<properties>
  <variables>
    <variable name="base_dir" value="${system:java.io.tmpdir}"/>
  </variables>
  <category name="general">
    <hello>world</hello>
    <text>
    Hello,
    this is me!
    </text>
    <directory>${base_dir}/docs/mydir</directory>
  </category>
  <category name="server">
    <port>8080</port>
    <docroot>${base_dir}/docs</docroot>
  </category>
</properties>
```

7.3 NestedConfigParser

The NestedConfigParser lets you defined categories inside categories.

This is an example that sums it up:

```
<properties>
  <variables>
    <variable name="base_dir" value="${system:java.io.tmpdir}"/>
  </variables>
  <category name="general">
    <server_port>8080</server_port>
    <server_root>${base_dir}/docs</server_root>
  </category>
  <category name="settings">
    <category name="client">
      <screen_size>1024,768</screen_size>
    </category>
    <category name="server">
      <home_dir>${base_dir}/app</home_dir>
    </category>
  </category>
</properties>
```

There is a demo called NestedConfigDemo included in the demo subdirectory.

7.4 Defining which parser to use

There are a few ways how you can tell jConfig which parser should be used. The best way is to include a file called jconfig.properties in the classpath and define the parser inside. You can define the default parser that will be used by all configurations are a specific parser for a specific configuration. Inside the file put a line:

```
# this defines the default parser and this is the good old parser
parser=org.jconfig.parser.DefaultConfigParser
```

In order to define a parser for a specific configuration put the name after the parser:

```
# this defines a parser for the configuration called "myConfig"
parser.myConfig=org.jconfig.parser.NestedConfigParser
```

If you do not want to include a jconfig.properties file then you can set a system property called "jconfig.properties" and define the full qualified class name of the parser:

```
System.setProperty("jconfig.parser","org.jconfig.parser.CDataConfigParser");
```

This way you cannot specify different parsers for different configurations.

7.5 Writing your own parser

Since jConfig supports pluggable parsers you can write your own. For example if you want to read your configuration from a database. Your parser has to implement only one method:

```
public Configuration parse(Document doc,String configName);
```

The parser is called from the handler and receives the document and the configName. The method must return the configuration. If you want to you can also implement your own configuration.

8 Error handling

jConfig is designed in a way that it will never crash your application. The error messages and exception are kept internal. This can be a downfall if you are facing strange behaviour. When you are sure you have the configuration file in the classpath but you get an empty configuration from jConfig then it was hard to find the error.
This new version now includes an ErrorHandler that you can switch on to find any problems.
The default error handler will not report any errors. There is a error handler included that will write out any message or exception to the console.

There are two ways of setting the ErrorHandler.

1. system property
You set a system property with the error handler:

```
System.setProperty("jconfig.errorhandler","org.jconfig.error.SimpleErrorHandler");
```

2. put an entry in the jconfig.properties
Create a file called jconfig.properties and put in the following line:

```
jconfig.errorhandler=org.jconfig.error.SimpleErrorHandler
```

If none of the above is set then the default NullDeviceErrorHandler is used.

You can also write your own error handler. All you need is to implement the org.jconfig.error.ErrorHandler interface and define your error handler with the full qualified class name.

Example:

```
package org.foo.myapp.config;

public class MySimpleErrorHandler implements ErrorHandler {

  public MySimpleErrorHandler() {
  }

  public void reportError(String message) {
    System.out.println(message);
  }


  public void reportError(String message, Throwable thrown) {
    // do something here like send an email or call someone
    System.out.println(message);
    thrown.printStackTrace();
  }
}
```

and then define it like this:

```
jconfig.errorhandler=org.foo.myapp.config.MySimpleErrorHandler
```

NOTE: As of version 2.8 the SimpleErrorHandler is now the default. This means all errors are written to the console.

9 MBeans – Using JMX

jConfig offers a few MBeans. There are two JBoss specific MBeans available and two generic Mbean that can used in any JMX enabled environment.

9.1 In general

The jConfig MBean lets you configure your configurations inside JBoss. There is also a method which you can use to reload a configuration manually if the file has changed and you are not using the auto-reload feature of jConfig1.3.

9.2 The first generic MBean

This simple MBean lets you load a configuration with a defined configuration name. It also supports methods for setting a property or removing a property. You can also view the configuration and reload and save it.

9.2.1 Installation

The installation requires a few simple steps. This part describes how to deploy it in a JBoss2.x server.

- copy the jconfig.jar to the $JBOSS_HOME/lib/ext directory.
- put the following lines in the $JBOSS_HOME/conf/jboss/jboss.jcml (or where ever your jboss.jcml is)

```
<mbean code="org.jconfig.jmx.ConfigHandler" name=":service=Configuration,name=MyConfig">
  <attribute name="ResourceName">config.xml</attribute>
  <attribute name="ConfigurationName">MyConfig</attribute>
</mbean>
```

Copy your config.xml (or whatever name you have chosen for your configuration file) to the $JBOSS_HOME/conf/jboss directory. Repeat step (3) and (4) for all the different configurations that you need but do not forget to specify a different name and a different file. That's all. The attributes are explained in the next section.
Note: Both the resource name and the configuration name must be set
NOTE: If you run JBoss together with tomcat then the conf-directory is $JBOSS_HOME/conf/tomcat or if you use Tomcat4 (catalina) then it is $JBOSS_HOME/conf/catalina

9.2.2 Configuring the MBean

We will now explain the attributes in more detail.
ResourceName
This attribute defines the configuration file. It does not have to be a file. You can also use "http://www.myserver.com/configs/myconfig.xml".
ConfigurationName
This is the name of your configuration instance. If you choose for example "MyConfig" then you can use the ConfigurationManager in this way:

```
private static final ConfigurationManager cm = ConfigurationManager.getInstance
("MyConfig");
```

9.3 The generic dynamic MBean

Beside the two JBoss specific MBeans there is also one generic MBean included. This does not implement any of the JBoss interfaces.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE server>

<server>

  <classpath codebase="lib" archives="jconfig.jar"/>
  <mbean code="org.jconfig.jmx.ConfigurationManagerDynamicMBean"
name="user:configuration=DynamicConfiguration">
  </mbean>

</server>
```