org.jax.mgi.shr.config Design Document

Author: Mike Walker

Created: March 19, 2003

Last Modified: September 16, 2004 23:47

1 Purpose of Document

This document describes the classes belonging to the org.jax.mgi.shr.config package and provides source code examples for common usage patterns.

2 Introduction

The org.jax.mgi.shr.config package is comprised of three central java classes used primarily for reading, storing and accessing configuration parameters within configuration files and java system properties. They are **Configuration**, **ConfigurationManager** and **Configurator**. There are other classes belonging to org.jax.mgi.shr.config package that may come from separate products and are intended to configure classes from other packages. The methods for accessing configuration parameters are protected, and any class developed to extend the core classes must be written as part of the org.jax.mgi.shr.config package. These extended classes provide methods for looking up parameters within a logical grouping of configuration information. An example of this type of class is the **BCPManagerCfg** class which belongs to the config package but configures the **BCPManager** class from the org.jax.mgi.shr.dbutils.bcp package. This class is responsible for looking up configuration parameters pertaining to BCP processing. Typically, by convention, these parameters which belong to a logical grouping are prefixed in a consistent way. For example, the configuration parameters read by the **BCPManagerCfg** class all begin with the string 'BCP' (BCP_PATH, BCP_DELIMITER). Section 6 discusses these types of classes in more detail.

This product integrates with the org.jax.mgi.shr.exception package and the org.jax.mgi.shr.types package.

It has been coded and tested with Java 1.4.1 JDK.

3 Overview of Classes

The **Configuration** class is used to represent a configuration file or a set of configuration files. It can read a configuration file in any of the three formats, Bourne shell, csh shell and java properties format (name=value). This class is used by the **ConfigurationManager** class and is typically not instantiated directly.

The **ConfigurationManager** class manages multiple configuration files and uses the Configuration class to read and internally store the configuration information. This class follows the singleton design pattern and therefore only one instance of the class ever exists at a time. You can have

multiple references to the object, but only one instance within the JVM. The constructor for this class is private and a call to the getInstance() method is used to obtain the singleton reference.

The **Configurator** class is a base class that has an instance of a **ConfigurationManager** and provides methods for accessing configuration information within a **ConfigurationManager.** Furthermore, the **Configurator** has methods for converting String results read from the file to their proper types and also for handling the parameter-not-found exception in various predefined ways. A class that is interested in providing access to a logical grouping of configuration parameters would extend this class and provide lookup methods for each of the parameters. Section 6 discusses the use of this object.

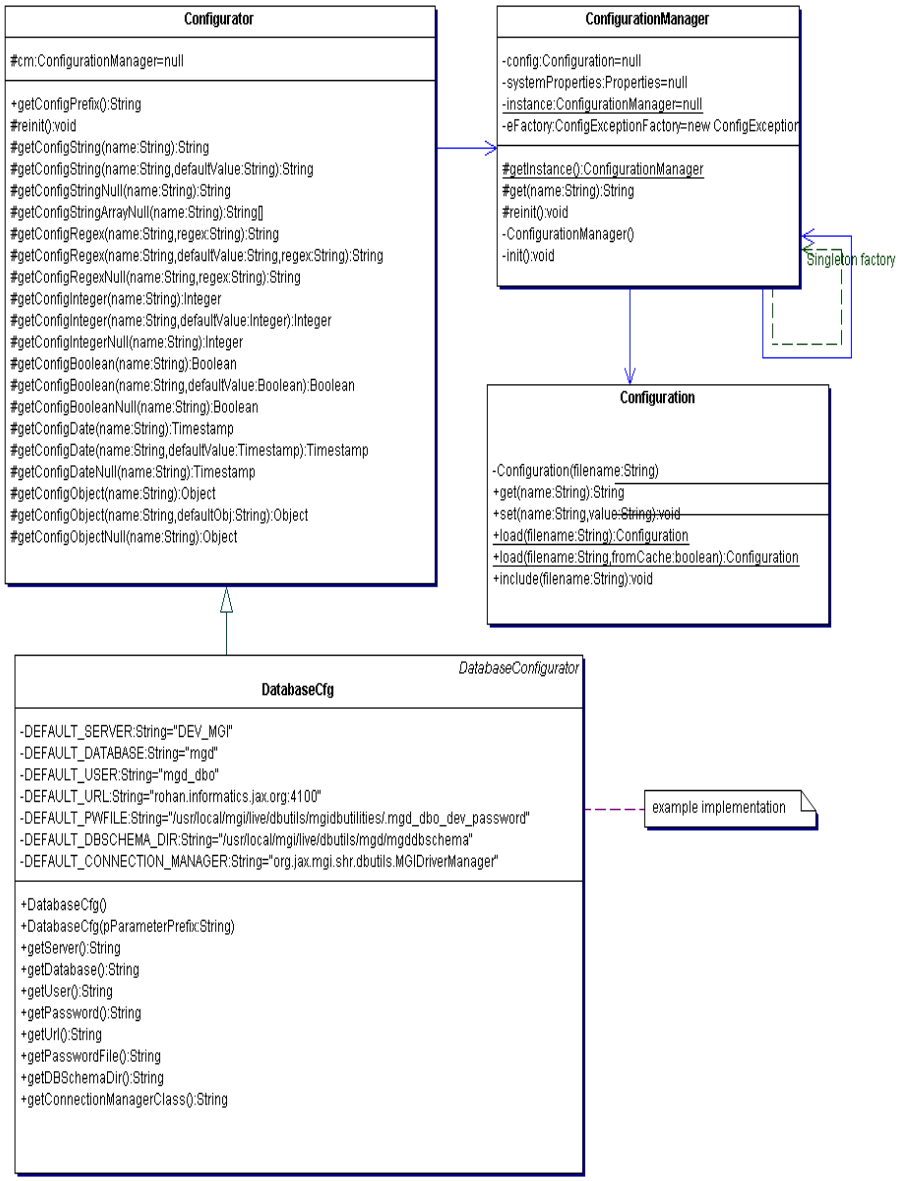
The **RebelReader** class has an instance of a **ConfigurationManager** class and provides a public get method for looking up any parameter value by name. The get methods of the **ConfigurationManager** class are protected. By using a **RebelReader**, any application can lookup a parameter in the configuration file without writing or using a specific **Configurator** class as discussed in Section 6. This may be convenient is some cases, but entails hardcoding parameter values in your application code. One responsibility of the extended **Configurator** objects is to hardcode the parameter names within their lookup methods, but these classes always exist in the org.jax.mgi.shr.config package, and therefore provides a central place for storing hardcoded values, and simplifies longterm maintenance of system configuration access.

The **ConfigReinitializer** class has one static method for rereading configuration parameters from the configuration files and java system properties. Once the parameters have been read, they are static and stored internally in cache. If the system properties or config files change during the runtime of the system, this class can be used to reinitialize the internal configuration cache.

The **ConfigException** extends **MGIException** from the org.jax.mgi.shr.exception package. It is the primary means of propagating exceptions out of the configuration frameworks classes. Each **ConfigException** contains an error message pertaining to an error event and the **MGIException** class provides a way to bind runtime values in a dynamic way to predefined messages.

The **ConfigExceptionFactory** class stores named instances of **ConfigExceptions** within an internal cache and provides access to these exceptions by name. The frameworks classes are responsible for knowing the names of these exceptions, accessing them through the **ConfigExceptionFactory**, binding runtime information to the error messages, and throwing them during a system fault.

4 Class Diagram



5 Configuring an Application

The **ConfigurationManager** class reads the CONFIG system property to obtain a list of configuration files. This property takes a comma separated list of filenames as its value. A **Configuration** object is created internal to the **ConfigurationManager** class which reads each configuration file. The get method of the **ConfigurationManager** calls the get methods for each **Configuration** object. The filenames which come last in the list have precedence over those that are antecedent.

Also command line system properties have precedence over the values from the configuration files. See Example 1 for setting up the configuration for an application.

EXAMPLE 1. Configuring an application

```
java -DCONFIG=file1,file2 -DPARAM1=VALUE1 -DPARAM2=VALUE2 <app>
```

6 Configurator Classes

The get methods of the **ConfigurationManager** class are protected and are intended to be used only by members of the org.jax.mgi.shr.config package. In order to access configuration parameters, a typical pattern is to create a new class that belongs to this package which extends **Configurator,** and therefore has a reference to the **ConfigurationManager**. This class is referred to as a "configurator" class. This base class provides lookup methods for easy access to the configuration parameters. The super class knows the names of configuration parameters that belong to a selected logical grouping and provides public accessor methods which calls the base class lookup methods with these names as parameters. See Example 2.

EXAMPLE 2. A configurator class

```
public class SomeConfigurator extends Configurator
{
  public SomeConfigurator() throws ConfigException
  {
    super();
  }
  public String getSomeParameter()
  {
    super.getConfigString("HARDCODED_PARAMETER_NAME");
  }
}
```

The Configurator base class provides methods for interpreting lookup values as either strings, integers or boolean. These are getConfigString(), getConfigInt() and getConfigBoolean(). The expected range of boolean values allowed within the configuration file includes the case insensitive strings 'true', 'false', '1', '0',' yes' or 'no'. A ConfigException will be thrown if a conversion error occurs while executing one of these methods. There are additionally get methods that provide different behaviors for the parameter-not-found event. One method will throw an error. Another will return null, while another will return a default value which is provided as a parameter to the get call (see example 3, 4, 5).

EXAMPLE 3. Throwing errors when parameter is not found

```
public String getSomeParameter()
throws ConfigException
{
  return super.getConfigString("HARDCODED_PARAMETER_NAME");
}
```

EXAMPLE 4. Providing default value when parameter is not found

```
public boolean getSomeBooleanParameter()
throws ConfigException
{
  return super.getConfigBoolean(
       "HARDCODED_PARAMETER_NAME", true);
}
```

EXAMPLE 5. Returning no exception on parameter-not-found

```
public boolean getSomeParameter()
{
  return super.getConfigStringNull("HARDCODED_PARAMETER_NAME");
}
```

The following table summarizes the methods provided in the Configurator base class and what happens when the lookup finds no value for the given parameter.

Table 1: Configurator methods

method	key not found behavior
String getConfigString(String)	throws ConfigException
String getConfigStringNull(String)	returns null
String getConfigString(String, String)	returns arg 2 as the default value
Integer getInteger(String)	throws ConfigException
Integer getIntegerNull(String)	returns null
Integer getInteger(String, String)	returns arg 2 as the default value
Boolean getBoolean(String)	throws ConfigException
Boolean getBooleanNull(String)	returns null
Boolean getBoolean(String, String)	returns arg 2 as the default value
Timestamp getConfigDate(String)	throws ConfigException
Timestamp getConfigDateNull(String)	returns null
Timestamp getConfigDate(String, String)	returns arg 2 as the default value
Object getObject(String)	throws ConfigException
Object getObjectNull(String)	returns null
Object getObject(String, String)	returns arg 2 as the default value

The following tables provides a list of classes currently extending the **Configurator** class within the lib_java_core product and shows which classes use them and from which package the client classes come from.

Table 2: list of configurator classes

config class	client class(es)	client package(s)
BCPManagerCfg	BCPManager	org.jax.mgi.shr.dbutils.bcp
BCPWriterCfg	BCPWriter	org.jax.mgi.shr.dbutils.bcp
DatabaseCfg	SQLDataManager	org.jax.mgi.shr.dbutils
InputDataCfg	InputDataFile	org.jax.mgi.shr.ioutils
LogCfg	any class that uses a logger	numerous
RecordStampCfg	RecordStamper	org.jax.mgi.shr.dbutils
ScriptWriterCfg	ScriptWriter	org.jax.mgi.shr.dbutils
CacheCfg	RowDataCacheHandler	org.jax.mgi.cache