Jalisto

User Manual

Table of Contents

Table of Contents 2
Jalisto Overview 4
1.1 Overview 4
1.2 Getting started 4
1.3 User modes 5
1.4 Jalisto modules 6
1.4.1 Query engine Module 6
1.4.2 JMX administration Module 6
1.4.3 JCA Module 6
1.5 Storage solutions 6
2 Jalisto core features and configuration 8
2.1 Required libraries 8
2.2 Configuring Jalisto 8
2.2.1 Core module options 8
2.2.2 Configuration file sample: 9
2.3 JalistoFactory, Session and Transaction 9
2.3.1 Jalisto Factory 9
2.3.2 Session 10
2.3.3 Transaction 10
2.3.4 Code sample 10
2.4 Meta definition 10
2.4.1 Define a class 10
2.4.2 Schema evolution 12
2.5 Persistent instance to array conversion 12
2.6 Dealing with persistent instances 13
2.7 Jalisto OID 14
2.8 User modes 14
2.8.1 Mono 14
2.8.2 Multi 14
2.8.3 Read only 15
3 Client-server functionalities 16
3.1 Client’s point of view 16
3.2 Server’s point of view 16
4 The storage layers 18
4.1 Storage layer mechanism and configuration 18
4.2 RAF layers 18
4.3 RAF-nolog implementations 18
4.4 RAF-log implementations 19
4.5 Memory implementation 20
4.6 How to choose and write a storage implementation 20
5 Admin tools and miscellaneous features 21
5.1 Trace system 21
5.2 Reorganize the datastore 21

5.3 Storage recovery ... 21
5.4 Browse the datastore ... 22
6 Query module ... 23
6.1 Overview of query module features ... 23
6.2 How to build a query ... 23
6.3 Execute a query ... 24
6.4 Indexes ... 24
7 JMX Administration ... 26
7.1 MBeans ... 26
7.1.1 CacheAdmin ... 26
7.1.2 ClassDescriptionAdmin ... 26
7.1.3 MemoryAdmin ... 27
7.2 Run the Jalisto JMX server ... 27
7.3 Example ... 28
8 Jalisto internal design aspects ... 29
8.1 The Jalisto page system ... 29
8.2 Cache ... 30
8.2.1 Page cache ... 30
8.2.2 New cache implementation ... 31
8.2.3 Cache options ... 31
8.3 Customized RandomAccessFile architecture ... 31
8.3.1 The file headers region ... 32
8.3.2 The index region ... 32
8.3.3 Record data region ... 33
9 Annex ... 34
9.1 Jalisto configuration properties summary ... 34

Jalisto Overview

1.1 Overview

Jalisto stands for JAva LIgth STOrage and is a modular low-level object datastore entirely written in java.

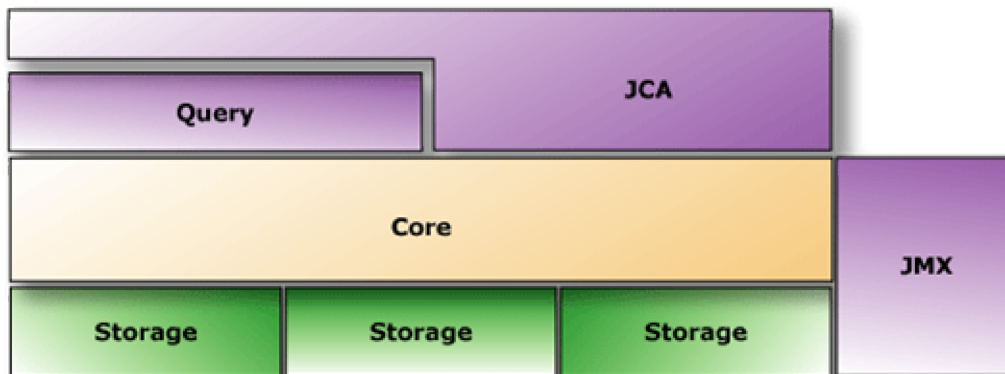
Jalisto is better when used in combination with a high level API (JDO or SDO) implementation such as Xcalia Intermediation Core™.

Jalisto can be used in embedded applications running into mobile device with low limited memory but it can also be used by any application that needs a simple, pure Java and efficient data storage for local information (user profiles, non-structured information, logging and traces...).

Jalisto is build around a core system. The storage layer can be switched, caches can be plugged, and optional modules are available.
There are currently three available optional modules:

- a query engine,
- a JMX administration framework,
- a JCA compliant layer.

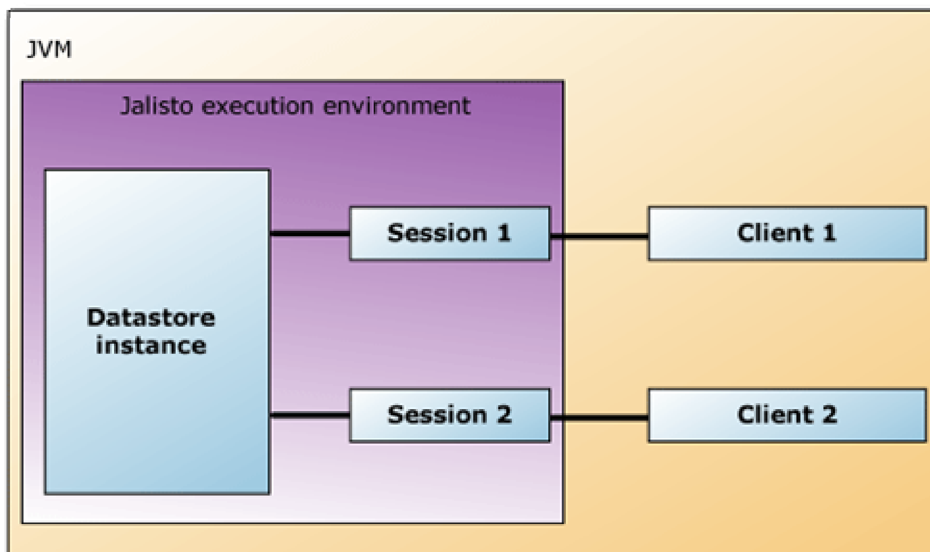
The global architecture is summarized below:



1.2 Getting started

A datastore instance is configured by a single properties file. A user who wants to work on a specific datastore instance must specify the properties file of this instance.
This file contains datastore's files paths, datastore's user mode, and custom Jalisto properties.

Jalisto users will mainly use JalistoFactory, Session and Transaction instances.
Session is the main Jalisto interface. It allows creating, retrieving, updating and deleting objects (known as CRUD operations). Such a session is always user-specific.



A working session is obtained from the JalistoFactory, using the configuration file path as parameter. This path may be an absolute path, a path relative to the execution directory, or the name of a file available from the classpath.
An FdbTransaction instance, obtained from the session, will allow the user to demarcate transactions boundaries around his operations.

1.3 User modes

Jalisto can be use in a mono-user mode, i.e. one client using a unique session on a single Jalisto instance in a local JVM.
With the multi-user mode, multiple threads in a local JVM can connect on a single Jalisto instance, using multiple concurrent sessions.
The read-only user mode allows multiple users in a single JVM to read (and only read) objects. No locks are used, no logging is used, no updates can be committed. This version is lighter than the two others, and obviously much faster too.
The user mode used to create the datastore instance has no impact on how the datastore can be used later.

A datastore instance cannot be opened by more than a single Java process. The client-server architecture allows multiple distant clients to access and work with a remote Jalisto instance. The server side Java process opens the datastore and works with it.

The user mode is specified by the “userMode” property in the instance properties file. Values can be either “mono”, “multi” or “readonly”. See the specific chapter for more detail about Client-sever mode.

1.4 Jalisto modules

Three optional modules can be used with the Jalisto core system. To read more information about these optional modules, see specific chapters.

The module’s jar must be added in the runtime classpath to activate its features.

1.4.1 Query engine Module

The query engine allows executing queries, and maintaining indices on persistent classes’ fields. To deal with Jalisto queries, a query manager is associated to each Jalisto session. This module associated jar file is “jalisto-query.jar”.

1.4.2 JMX administration Module

This module implements a set of JMX (JSR 3) compliant MBeans to manage and monitor Jalisto configuration and runtime parameters such as:

- Defined persistent classes’ schemas,
- Internal caches size,
- Runtime memory size.

For more information on JMX, JSR 3 and JSR 160 visit http://java.sun.com/products/JavaManagement/

1.4.3 JCA Module

This layer allows high level software to use Jalisto as a JCA connector.
NB: The JCA module is still under development and requires additional development and documentation.

1.5 Storage solutions

Jalisto can use different storage layers. The storage strategy must be selected during the datastore creation. Once created, this property cannot be changed later.
Writing a new storage layer is possible. A new class implementing org.objectweb.jalisto.se.api.physical.PluggablePhysicalFileAccess interface must be supplied.

The “physicalClass” property in the configuration file must be set in order to specify the storage implementation class. If nothing is specified, the default storage solution is used, which is the “RAF nolog” synchronous implementation.

Available storage strategies are:

- Memory: this implementation keeps all persistent data in memory. No files are created.
- Random Access File (RAF): this implementation uses a customized Java RandomAccessFile to store data. The random access File strategy has different flavors:
    - Data can be stored within a single file or within multiple files.
    - I/Os can be synchronous or asynchronous.
    - An optional logging mechanism is available to support true ACID transactions.

2 Jalisto core features and configuration

2.1 *Required libraries*

Jalisto is composed of several jar files. These files must be added to the application classpath. The exact list of files required is dependent on user's needs.

The list of Jalisto jar files is described here under:

Module name	Description	Jar	External dependencies
Core		Jalisto-core.jar	
Core		Jalisto-core-mono.jar	
Core		Jalisto-core-tool.jar	
Core	Global test framework, core test suite	Jalisto-core-test.jar	Junit.jar, jfunc.jar, jcfe.jar
Query		Jalisto-query.jar	
JMX		Jalisto-jmx.jar	Mx4j.jar
JCA		Jalisto-jca.jar	Connector-1_0.jar
Storage	Memory storage	Jalisto-storage-memory.jar	
Storage	Random access file, no log	Jalisto-storage-raf.jar	
Storage	Random access file, synchronous, no log	Jalisto-storage-synraf.jar	
Storage	Random access file, log	Jalisto-storage-raflog.jar	

2.2 *Configuring Jalisto*

Jalisto configuration is done through a Java properties file. There is exactly one properties file associated with each datastore instance.
The configuration file format is the classical key=value format of Java properties files.

To create a datastore instance, the Jalisto core system needs to know some specific properties. These properties' values are specified in a specific properties file.
When the datastore instance is reopened later (during the instantiation of a Session), the same properties file helps the core system to locate the physical storage files on disk.

NB: all properties have default values, so only specific properties with none-default values have to be overwritten.

2.2.1 Core module options

Key	Description	Value type	Default

			value
name	Give a name to the datastore instance	String	“jalisto”
dbFilesPaths	Path(s) of database file(s). These paths are absolute, or relative to Jalisto execution directory	Comma-separated list	<datastore name>.jalisto
userMode	Selection between mono-user, read-only and multi user modes.	“mono”, “readonly”, “multi”	“mono”
trace	Enable logging for specified modules.	Comma-separated list	Empty list
physicalClass	The full class name of the chosen implementation of the PluggablePhysicalFileAccess interface.	String	See in chapter The storage layers
concurrencyMode	If the datastore is in multi-users mode, specifies an optimistic or pessimistic concurrency mode.	“optimistic” or “pessimistic”	“optimistic”
logFile	Path of transaction log file. This path is absolute, or relative to Jalisto execution directory	String	"<datastore name>-log.jalisto"
dbInitialSize	Datastore initialization size, in page count.	int	“1000”
oidTableSize	Number of ‘logical – physical’ links in oidTable.	int	“10000”

2.2.2 Configuration file sample:

```
name                              employees
userMode                          mono
dbFilesPaths                      employees.jalisto
instancePageSize                  100
physicalClass
       org.objectweb.jalisto.se.storage.raf.nolog.asynchro.PhysicalFileAccessNolog
AsynchroImpl
```

2.3 JalistoFactory, Session and Transaction

2.3.1 Jalisto Factory

The org.objectweb.jalisto.se.JalistoFactory is the only factory required. This factory allows to get a working session, and to define metadata. It also allows starting the JMX server, if the administration module is enabled.

2.3.2 Session

Session is the main Jalisto user interface to create, retrieve, update, and delete objects. Such a session is always user-specific.

To obtain a working session, the datastore instance properties file path must be supplied to the factory. This path may be either an absolute path, a path relative to the runtime directory or the name of a file available in the classpath.

The "openSession()" method must be called before any other Session methods calls. At the working session end, the "closeSession()" method must be explicitly called in order to clean all the system resources used by Jalisto.

2.3.3 Transaction

A Transaction instance, obtained from the Session, will allow the user to demarcate transactions boundaries around his operations, by calling the traditional begin and commit/rollback methods.
NB: all sessions, in all user modes, are transactional.

Read the Jalisto javadoc to have a complete list of all operations available from JalistoFactory and Session.

2.3.4 Code sample

```
Session session = JalistoFactory.getSession("jalisto.properties");
session.openSession();
Transaction tx = session.currentTransaction();

tx.begin();
…
tx.commit();

tx.begin();
…
tx.commit();

session.closeSession();
```

2.4 Meta definition

2.4.1 Define a class

The Jalisto core system needs the definition of the persistent classes.
All the methods needed to define and remove persistent classes are available from JalistoFactory and Session.

For each persistent class the following information must be defined:

- The class name
- For each field:

- The field name
- The field type

Jalisto supports most Java types such as Integer, String, Date, Collection and Map types (including Vector and HashMap), "link" type for reference to other persistent instances. As Jalisto is an object datastore, primitive Java type like "int" or "short" are not supported, and thus must be converted in their related object wrapper type.

All other types will be serialized, so must implement the Serializable interface.

Available Jalisto types are:

- Atomic types:
  - DoubleType
  - FloatType
  - IntegerType
  - LongType
  - ShortType
  - StringType.
- Reference types:
  - LinkType
  - SerializedType.
- Collection types:
  - ArrayType
  - CollectionType
  - MapType.

JalistoFactory is used to create ClassDescription and FieldDescription instances.
User has to create the ClassDescription with the persistent class fully qualified name, and to add fields to the class description. For each of them, specify the type name and Jalisto type. Then the Session is used to actually define the class within the datastore.

All metadata definitions must be done outside transaction boundaries, and no other concurrent transaction can be started in the meanwhile.
No field has to be defined for the Object unique IDentifier (OID).

```
ClassDescription meta =
        JalistoFactory.createClassDescription(Book.class.getName());

meta.addField(JalistoFactory.createFieldDescription("title",
                                                    new StringType()));
meta.addField(JalistoFactory.createFieldDescription("pages",
                                                    new IntegerType()));
meta.addField(JalistoFactory.createFieldDescription("price",
                                                    new IntegerType()));
meta.addField(JalistoFactory.createFieldDescription("author",
                                                    new LinkType()));
meta.addField(JalistoFactory.createFieldDescription("format",
                                                    new SerializedType()));

session.defineClass(meta);
```

The defined class can be deleted. Jalisto system first remove all instances of this class, and then remove the class definition.

```
session.removeClass(Book.class.getName());
```

2.4.2 Schema evolution

The schema evolution feature allows updating an already created schema, in order to reflect evolutions in persistent classes definition. Fields can be added, renamed and removed.
When adding a new field to a persistent class, the default field value is null for all existing class instances.

Schema evolution methods are available from MetaRepository, which can be find via the "getMetaRepository()" method on working Session. All the schema evolution functionalities must be called outside transaction boundaries. See the MetaRepository API to get all detailed informations.

2.5 Persistent instance to array conversion

As Jalisto is a low-level datastore, so there is no "user friendly" mapping layer in the core package.

Persistent object instances must be manually converted into a primitive form, according to their class meta-definition.
This primitive form is an object array, or "Object[]". That array contains all field values, in the order defined in the class meta-definition. Primitive values must be converted to their equivalent object wrappers ("int" becomes "Integer"). References to persistent instances must be replaced by the referenced object's LogicalOid instance. Others can remain the same, as long as they are Serializable.

Example:

```
public class Book {

    private String title;
    private String author;
    private int price;

    public Book() {
    }

...

    public Object[] toArray() {
        Object[] result = new Object[4];
        result[0] = title;
        result[1] = author;
```

```
        result[2] = new Integer(price);
        return result;
    }

    public static Book toBook(Object[] array) {
        Book book = new Book();
        book.setTitle((String) array[0]);
        book.setAuthor((String) array[1]);
        book.setPrice(((Integer) array[2]).intValue());
        return book;
    }
}
```

NB: The future release of Jalisto will implement automatic mapping mechanism to avoid this step.

2.6 Dealing with persistent instances

Once persistent classes are defined, and instances are converted into arrays, dealing with Jalisto is easy.
All database operations are executed within transaction boundaries.

The user can make data persistent in one or two steps:
Using the “one step” method, user gives the class and the values array, and gets back the identity of the new persistent instance.
Using the “two steps” method, user first allocates space in the datastore by giving the instance class, and then gets back the identity. He gives the values array and the identity to make the values persistent.

```
// persistent object in one step
Object oid = session.createObject(book.toArray(), Book.class);

// persistent object in two steps
Object oid = session.makeNewFileOid(Book.class);
...
session.createObject(oid, book.toArray());
```

To read a persistent instance in its array form, its identity is needed:

```
Object[] o = session.readObjectByOid(oid);
```

If the user doesn’t have the oid, he can read all persistent instances’ oids from a given class by using an extent. The “getExtent()” method return a LogicalOid collection:

```
Iterator extent = session.getExtent(aClass).iterator();
```

NB: if more functionalities are needed to retrieve objects, use query mechanisms (see the Query module).

To update a persistent instance, user needs its identity, and the new values array:

```
session.updateObjectByOid(oid, newArrayValue);
```

To delete a persistent instance, user just needs the instance's identity:

```
session.deleteObjectByOid(oid);
```

2.7 Jalisto OID

By making instances persistent, or using extents, user gets back Object unique IDentifier, or oid. The oid is build by the Jalisto internal system, it's an instance of the LogicalOid class. It's a wrapper around a long value.

The wrapped long value is composed of 2 parts. First, a value for the instance class, and second the instance number. When printed, the LogicalOid instance shows these two parts, with a String like "12.4751" which means "class number 12, instance number 4751".

2.8 User modes

The user mode is selected via the "userMode" properties in the configuration file. The "mono" value will run the mono user mode, the "multi" value the multi user mode, and "readonly" the read-only user mode (on an existing datastore instance). See the specific chapter for more detail about Client-sever mode.

The user mode used to create the datastore instance has no impact on how the datastore can be used later. A mono datastore instance can be used in a multi mode, and vice versa, or with the read-only mode. To do so, the only thing to do is to change the userMode property in the configuration file.
However, the same instance cannot be used *at the same time* with different modes.

2.8.1 Mono

Jalisto can be use in a mono-user mode, i.e. one client using a unique session on a single Jalisto instance in a local JVM.
If several calls in the JVM are made to the "getSession()" factory static method, the same Session instance will be returned.

2.8.2 Multi

With the multi-user mode, multiple threads in a local JVM can connect on a single Jalisto instance, using multiple concurrent sessions. Each call to the "getSession()" factory static method will return a different session.

Both pessimistic and optimistic concurrency modes are available. Jalisto will acquire locks on objects according to the concurrency mode selected.

Concurrency mode can be specified with the “concurrencyMode” option. Available values are “optimistic” and “pessimistic”. The default value is “optimistic”.

NB: whenever a deadlock is detected, a LockException is thrown.

2.8.3 Read only

Once a datastore is created and populated with other user modes, it can be accessed with the read only mode.
The read-only user mode allows several users in a single JVM to read (and only read) objects. No locks are managed, no logging is used, no updates can be committed. This version is lighter than the two others, and obviously much faster too.

3 Client-server functionalities

A datastore instance cannot be opened by more than a single Java process. The client-server architecture allows multiple distant clients to access and work with a remote Jalisto instance.

3.1 *Client's point of view*

From the client's point of view, working with a client-server Jalisto is exactly similar than working with a local Jalisto instance. Only the properties configuration file is different.

Example:

```
serverPropertiesFile   jalisto.properties
internalFactoryClass   org.objectweb.jalisto.se.impl.factory.InternalRemoteFactory
host                   localhost
port                   7777
communicationFactoryClass
org.objectweb.jalisto.se.impl.remote.socket.CommunicationFactorySocketImpl
```

The only mandatory property is 'internalFactoryClass', with the exact value org.objectweb.jalisto.se.impl.factory.InternalRemoteFactory.

The server properties file path is local to the server execution context.

Key	Description	Value type	Default value
internalFactoryClass'	Implementation class for the client Jalisto factory. Value MUST be set.	String	See above
serverPropertiesFile	Path to the server datastore configuration file.	String	See above
host	Server localization.	String	"localhost"
port	Communication port.	int	7777

3.2 *Server's point of view*

The server side's Java process opens the datastore and works with it.

A special launcher runs the server, which listen to specified port clients' requests.

A client starts by defining the datastore to use (with the corresponding properties file path). Then the server instantiate the datastore instance, manage local sessions, and reply to every further client's requests.

It's highly recommended to use multi-user mode for server side datastore instance, but read-only mode could be very useful to share information without writing permissions among remote users.

The ServerLauncher (in org.objectweb.jalisto.se.tool package) main method needs 2 arguments to start:
-port <int>: Port to listen. Clients connect to the server at this port. Default value is 7777.
-class <full class name>: Communication factory class. This factory instantiates implementation classes for the chosen communication protocol. Default value is org.objectweb.jalisto.se.impl.remote.socket.CommunicationFactorySocketImpl.

Port and communication factory class must be the same on both sides.

4 The storage layers

4.1 *Storage layer mechanism and configuration*

The storage is the underlying layer which physically reads and writes information on disk. Jalisto comes with a switchable storage layer, who allows working with different file format, storage strategy, and so on.

Jalisto can store information in a single file, or in several files, depending on the configuration. The physical storage file paths must be specified in the configuration file. The property is “dbFilesPaths”. The value is a comma-separated list of files’ paths. These paths can be absolute, or relative to the Jalisto configuration file directory.
The default path list is equal to “<datastore name>.jalisto”. See the configuration section for more information about Jalisto properties.

In the case of multiple files, the first one in the list will contain Jalisto system’s information. The second one will contain information about Object Identifiers. Then the next files will contain persistent instances.
Readers interested in the design of the Storage Layer should go to the Internal Jalisto Design section.

Examples:

```
dbFilesPaths  ../employees.jalisto

dbFilesPaths C:/dev/datatore/jalisto/jalisto-
sys.jalisto,C:/dev/datatore/jalisto/jalisto-oid.jalisto,D:/jalisto/jalisto-
inst01.jalisto,D:/jalisto/jalisto-inst01.jalisto,D:/jalisto/jalisto-
inst01.jalisto
```

4.2 *RAF layers*

RAF is the name for the main Jalisto storage layer implementations. They use a customized version of the standard java.io.RandomAccessFile.

An index had been added to the standard RAF, which maps a String key to a header. This header contains a physical address and the corresponding data’s length.

4.3 *RAF-nolog implementations*

These implementations are based upon the RAF layer. They don't log accesses to datastore, so transactions are not ACID, but these implementations are faster than logging implementations. So if logging is not needed, they could be used.

Two RAF-nolog implementations are available:
With the synchronous implementation, all disk writes are done before the system completes the commit or rollback.
With the asynchronous implementation, the system just enlists in a thread datastore operations that have to be written at commit time. This thread will *later* write in the datastore these enlisted operations. With this mode the commit process is faster, but at the end of the commit method call, all informations are not really persisted in the storage on the hard disk. This asynchronous implementation is "less ACID" than the synchronous one.

The implementation class for the raf-nolog-synchro mode is org.objectweb.jalisto.se.storage.raf.nolog.synchro.PhysicalFileAccessNologSynchroImpl.
This is the default implementation.
The implementation class for the raf-nolog-asynchro mode is "org.objectweb.fdb.storage.raf.nolog.asynchro.PhysicalFileAccessNologAsynchroImpl".

The classes for these nolog implementations are located in the "jalisto-storage-raf.jar" file. The jalisto-storage-synraf.jar file contains only the synchronous RAF nolog implementation.

4.4 RAF-log implementations

This implementation is also based upon the RAF layer. Logging functionalities have been added to the basic implementation, to complete ACID properties. If full logging functionalities are needed, we strongly recommend to use this implementation.

This layer logs (e.g. duplicates) information and operations in log files. Each persistence file has a log file, which allows recovering from unitary operations on the RAF file.
Each log file is at the same location and has the same name than the original file, but add "log" as postfix to the file name.

Operations done during the transaction are written in a transaction log file. So if the system crash during a transaction, it can cancel the entire transaction during the recovery process. This transaction log file can be specified within the configuration file with the property "logFile", giving the absolute or relative path (relative to configuration file's directory). The default value is "<datastore name>-log.jalisto".

The implementation class for the raf-log-synchro mode is "org.objectweb.jalisto.se.storage.raf.log.synraf.PhysicalFileAccessLogSynrafImpl".
The implementation class for the raf-log-asynchro mode is "org.objectweb.jalisto.se.storage.raf.log.asynraf.PhysicalFileAccessLogAsynrafImpl".

The implementation classes are located in the jalisto-storage-raflog.jar file.

4.5 Memory implementation

This implementation keeps all data in memory. No files are created on disk, and when the JVM exits, all information are lost.

The implementation class for the memory mode is "org.objectweb.jalisto.se.storage.memory.PhysicalFileAccessMemoryImpl".

The classes for this implementation are stored in the "jalisto-memory.jar" file.

4.6 How to choose and write a storage implementation

The implementation class is specified in the Jalisto configuration file, with the property "physicalClass". The fully qualified class name of a class which implements the PluggablePhysicalFileAccess interface must be given. The specified class is dynamically loaded at the execution, so the class must be in the Jalisto execution classpath.

User who wants to write a new storage layer just has to implement the same interface. All implementation classes must be in the Jalisto execution classpath at runtime. The configuration property "physicalClass" must be set with fully qualified class name of the root class of the new implementation.

The default value for the physicalClass property is "org.objectweb.jalisto.se.storage.raf.nolog.synchro.PhysicalFileAccessNologSynchroImpl", which is the implementation for the synchronous nolog RAF storage mode.

5 Admin tools and miscellaneous features

5.1 Trace system

Jalisto comes with a tracing system, which redirect its outputs to the standard output.
Those traces can be configured in the database configuration file, which contains a list of the modules to trace. If a module is activated, then all traces from that module will be displayed.
The trace property in the configuration file is called “trace”. The user must specify modules’ names he wants to trace in a comma-separated list.

Available modules:

Module name	Description
Session	Session instances.
Cache	Caches mechanisms
Oidtable	Logical/physical link table.
Logical	Allocation and logical management.
Physical	Database physical aspects.
Remote	Client-server aspects.
Debug	More verbose log, for debug purposes.

5.2 Reorganize the datastore

Deleting objects leaves empty “slots” in the datastore, which are re-used for new persistent objects. But sometimes, the datastore files size will stay larger than what really needed.
The reorganize feature defragments the storage files. Simply call the Session method “reorganize()” to use this feature.
NB: this method can only be called from a closed session. No other sessions must be active on the datastore.

5.3 Storage recovery

If system encounters a JVM crash, the datastore files may be corrupted. Recover from a corrupting crash is possible if a logging I/O implementation was used.
The revovery classes have a main() method, which must be called with the configuration file path as argument.
NB: it’s highly recommended to save datastore files before trying to recover them.

RecoverBase classes can be found in a chosen implementation sub-package, called “recover”. So, for the log synchronous implementation, the complete class name is “org.objectweb.jalisto.se.storage.raf.log.synraf.recover.RecoverBase”. For the

asynchronous implementation the revorery class is "org.objectweb.jalisto.se.storage.raf.log.asynraf.recover.RecoverBase"

5.4 Browse the datastore

Jalisto comes with a Datastore viewer tool. This tool display metadata and persistent instances stored in a given datastore.

The org.objectweb.jalisto.se.tool.JalistoViewer class can be found in the jalisto-core-tool.jar file. User just has to call the main() method of this class.

Options are available:

- only display metadata,
- show all instances,
- show only specified instances,

...

Call the main() method without argument to print help.

RAF storage implementations provide a lower-level viewer. It works only if there are not multiple storage files. This viewer displays the low–level customized random access file. User can see the Jaslito system's persistent objects, the pages, the oids, ... This is a very technical view of the storage.
User has to call the main() method of org.objectweb.jalisto.se.storage.RecordsFileInspector class. The configuration file path is the only argument to give.

6 Query module

Jalisto-query is an independent module which uses the Jalisto core system.

The query module goal is to allow users to create and execute queries to access persistent data. If user's project doesn't need more access facility than those already available in core (extent mechanism to get the oids), this module can be kept away from the classpath and minimize the footprint in memory.

All Jalisto query classes can be found in jalisto-query.jar file. This jar file must be present in the application execution classpath. If not, an exception will be raised when the Jalisto system will dynamically load the query manager class.

6.1 Overview of query module features

Users have to instantiate a new query manager from the Session. This query manager is the factory to build queries.
The Jalisto query system proposes an object-oriented query framework for Java that is based on APIs instead of declarative strings.

The programmer adds constraints to that query object using the query API, which allows navigation on the class's fields.
Parameters can be defined on the query. Their values have to be completed before execution.
Query's execution gives back to the user a set of results.

Indexes can be defined on persistent class fields to speed queries' execution, by using an index manager from the query manager.

6.2 How to build a query

The query manager is a factory for new queries. It is available from the current Jalisto working Session.
Once the query is build, it must be constrained with a persistent class. The query will be executed only on instances of this persistent class.
More constraints can be put on fields, using navigation with the "descend" method. With these constraints, a binary tree is build. All these operations could be done outside transaction boundaries.
See the APIs for a complete feature list.

Example:

```
QueryManager queryManager = session.getQueryManager();
```

```
Query query = queryManager.query();
query.constrain(Book.class);
Constraint c1 = query.descend("title").constrain("the hours").equal();
Constraint c2 = query.descend("price").constrain(new Integer(15)).equal();
c1.and(c2);
```

Navigation can be used to create constraints on a linked instance fields. To do so, the "descend" method must be called several times. An error in the navigation will be detected at execution time.
Parameters can also be used. The values of these parameters have to be filled just before execution.

```
QueryParameter titleParameter = new QueryParameter("title");
QueryParameter lastNameParameter = new QueryParameter("lastName");

Constraint c1 = query.descend("title").constrain(titleParameter).equal();
Constraint c2 = query.descend("author").descend("lastName").
        constrain(lastNameParameter).equal();
Constraint c3 = query.descend("price").constrain(new Integer(15)).lower();
(c1.or(c2)).and(c3);
```

6.3 Execute a query

Before query execution, a transaction must be started.
The "execute()" query method must be called, which returns a collection, basically an Iterator. This iterator contains Jalisto QueryResultWrappers instances. Each of these wrappers contains the identity and value of a valid query result.

```
tx.begin();
ObjectSet results = query.execute();
while (results.hasNext()) {
    QueryResultWrapper resultWrapper = (QueryResultWrapper) results.next();
    LogicalOid loid = resultWrapper.getLogicalOid ();
    Object[] value = resultWrapper.getValue();
}
tx.commit();
```

If query parameters are used, values must be bound to them. The binding operation can be done outside of transaction boundaries.

```
query.bind("title", "the hours");
query.bind("lastName", "Cunningham");
```

6.4 Indexes

Indexes can be built on any persistent class field. Indexes store a field instances' values, so execute a query with a filter on an indexed field is a lot faster than on a regular field.

An index manager is needed to define indexes. The index manager is available from the current query manager. The meta-description of the persistent class will be needed too, which are available from the MetaRepository (See Session API).
Then, inside transaction boundaries, an index can be build, giving the index of the field.

```
IndexManager indexManager = queryManager.getIndexManager();
ClassDescription meta =
        session.getMetaRepository().getMetaData(Book.class.getName());

tx.begin();
Index index = indexManager.buildIndexOnField(meta, meta.getIndex("title"));
tx.commit();
```

Indexes have a BTree structure, with nodes and leaves. Theses nodes and leaves are stored in storage pages. For each index, on each field, the node size can be customized, as the page size where nodes and leaves will be stored.
For instance, if an index on employee's first name is build, there will be a lot a data associated to each first name. Reduce the leaf page size will speed up the execution.
If an index on a key field is build, with a unique instance for each value, it will be better to increase the leaf page size.
The node page size, and the node size in the internal BTree structure, can also be adjusted.

All these customizations must be done before index creation in meta-description of the class.

```
ClassDescription meta =
session.getMetaRepository().getMetaData(aClass.getName());
IndexDescription indexDescription =
        meta.getFieldDescription(meta.getIndex(fieldName)).getIndexDescription();
indexDescription.setNodeSize((short)6);
indexDescription.setNodePageSize((short)20);
indexDescription.setLeafPageSize((short)50);
```

7 JMX Administration

Jaslito-jmx is an independent module which uses Jalisto core system, based upon MX4J implementation.
The JMX module goal is to provide administration MBeans to help users to configure, monitor and manage Jalisto datastore instances.

All JMX module classes can be found in jalisto-jmx.jar file. This jar file must be present in the application execution classpath. If not, an exception will be raised when user will try to start the JMX server. The MX4J jars must also be available in the classpath.

See the following reference URLs:
http://java.sun.com/products/JavaManagement/
http://mx4j.sourceforge.net/

7.1 MBeans

All Jalisto MBeans are dynamic and are independent from the chosen JMX implementation. They extend the abstract class JalistoAbstractDynamicMBean. This class helps programmers to implement new dynamic MBeans.

Jalisto MBean server, which launches the Jalisto MBeans, implements the interface MBeanServer.
To support a new JMX implementation, a new MBeanServer implementation has to be written and added to the JalistoMbeanServerFactory.

If new MBeans are written, they will need to be added in each server implementation launch process.

7.1.1 CacheAdmin

This MBean allows administrators to monitor Jalisto caches' size.
It displays the caches' size in real time, and administrators can adjust their maximum size.
Administrators can also switch the "keepInMemory" property, which tells if the read pages are cleaned at the transaction commit/rollback.

7.1.2 ClassDescriptionAdmin

This MBean helps administrators to define, read, modify and delete metadatas. It provides methods to define or remove a class description, and add or remove fields from a given description.

7.1.3 MemoryAdmin

This MBean only has read values, to trace classical Java runtime attributes: the free memory size and the total memory size. This MBean also returns the used memory size, which is equals to: total memory – free memory.

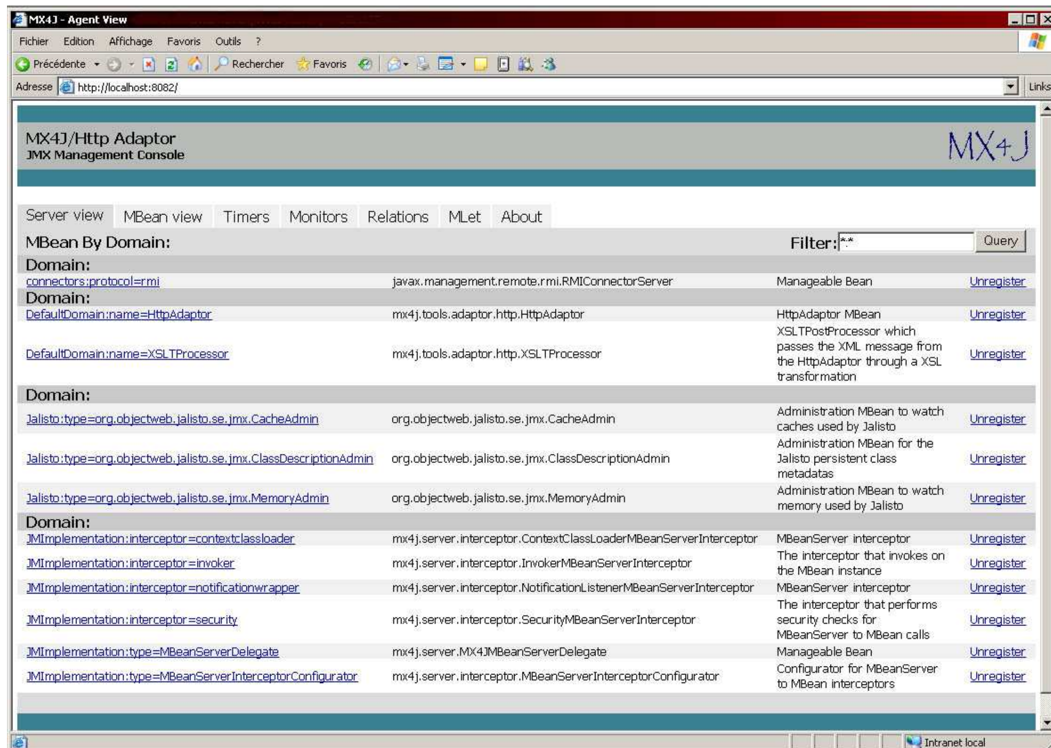
7.2 *Run the Jalisto JMX server*

To use functionalities provided by Jalisto JMX framework, the internal JMX server must be launch.
To do so, the "launchJMXServer()" method from the JalistoFactory must be called. Use the session or the configuration path as method parameter to indicate which datastore to manage.

The distribution also includes a test server used to demonstrate Jalisto administration capabilities. To launch this test server:

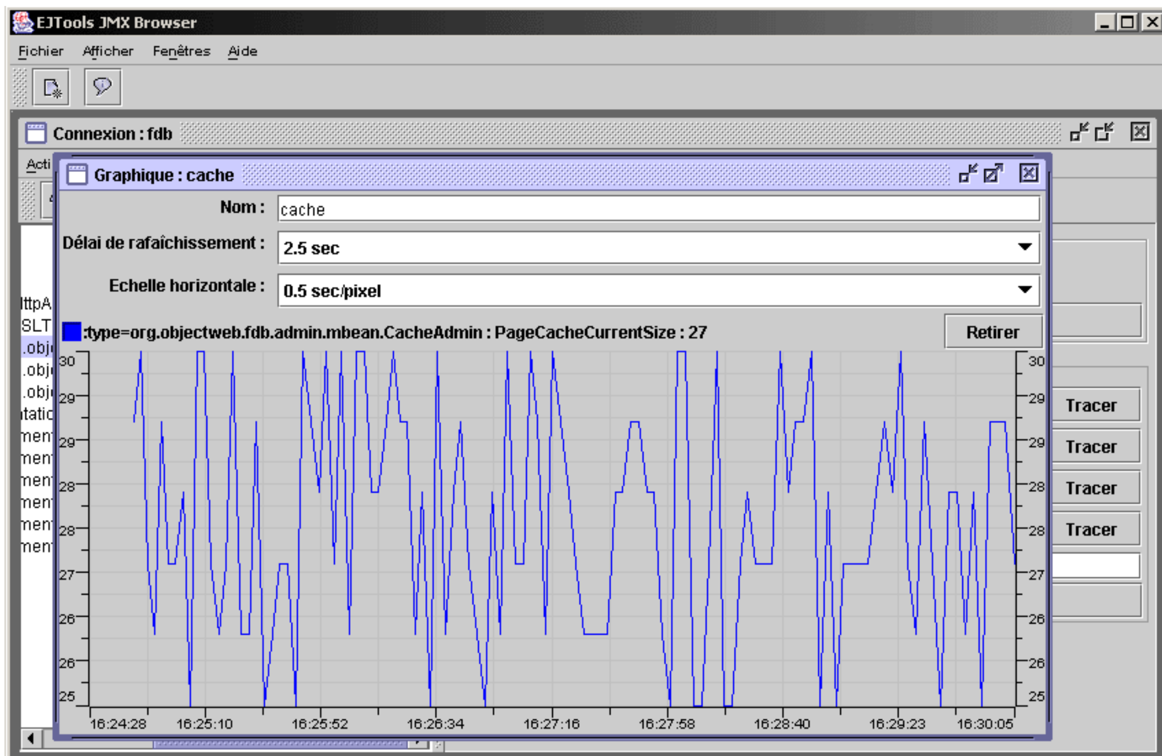
```
cd jmx
ant run.test-server
```

Then use your favorite web browser and connect to http://localhost:8082 to display Jalisto JMX console:



7.3 Example

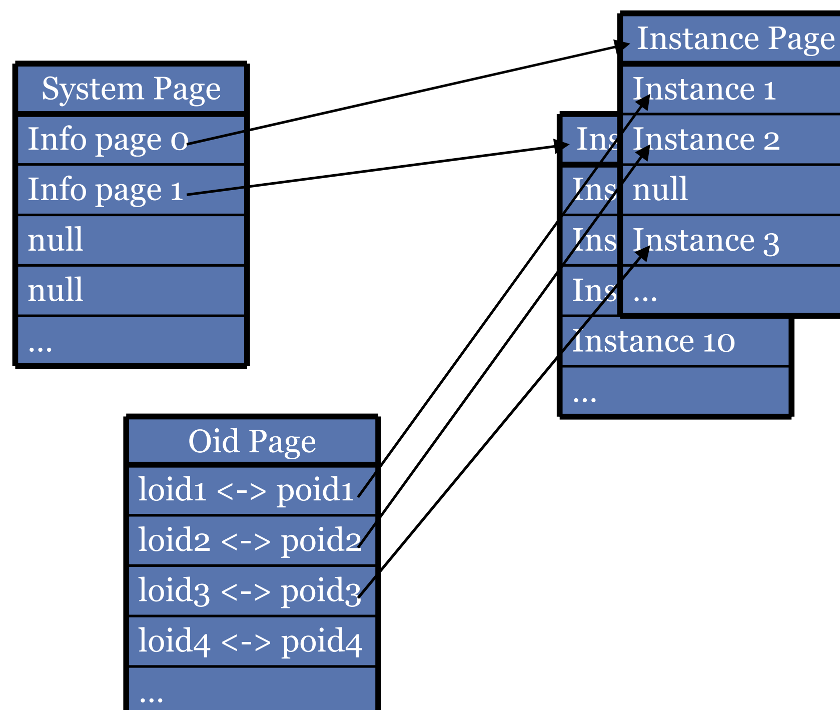
Once Jalisto JMX server is launched, it's possible to use tool to graphically edit the Mbean's properties.
In this example, we use EJTools (www.ejtools.org) to trace the page cache's size:



8 Jalisto internal design aspects

8.1 The Jalisto page system

Jalisto stores objects by groups instead of individually. Those groups are constituted depending on the type of data to write, and on their number. Those groups are called 'pages'.
When an instance is read from the database, the whole page that contains this instance is read at the same time. Consequently, a « neighbor » instance will be read more quickly in the same transaction.
In a similar way, when a persistent instance is modified, the whole page where it resides will be written into the database.



At a first glance this system can seem costly, as this page-granularity could be considered too coarse-grained. In fact, modifying a single element in a page is quite rare. Experiences based on various benchmarks proved using such a page-based system dramatically increase the overall performance.
Furthermore, the user can customize the page size, to tailor the database to the actual application needs. However, default values are those which resulted in best performance in our benchmarks, for a typical persistent data model.

Four page types exist in Jalisto:
« Instances pages » only contain persistent instances. If user's transactions work with only a limited number of instances, it is recommended to reduce the size of those pages.
The "logical-to-physical link pages" (or 'OID' pages) contain the links between logical object identities and their physical address into the datastore. These pages are read very often.
The "metadata pages" contain information about the persistent classes into the datastore. They are read in memory during the datastore instantiation, and kept there during the whole execution runtime.
The "system pages" contain internal informations about other pages allocations. They are read quite rarely, because most of these informations will be kept in memory once read.

Each page size can be configured in the Jalisto database configuration file. This size is equal to the number of data the page can contain. Consequently, a small size is recommended for the two later page types.

Key	Description	Value type	Default value
dbInitialSize	Datastore initialization size, in page count.	int	"1000"
oidTableSize	Number of 'logical – physical' links in oidTable.	int	"10000"
oidPageSize	Number of 'logical – physical' links per page.	int	"50"
classPageSize	Number of metadatas per page.	int	"10"
instancePageSize	Number of instances per page.	int	"200"
pageCacheSize	Transactional readed page cache size, in page count.	int	"20"
systemPageSize	Number of system informations per page.	int	"10"

8.2 Cache

Jalisto uses a low level cache for data pages, and a high level cache for object.

The current cache implementation is a LRU cache. Objects are stored within a LRU cache, to speed up further access. If the number of objects grows up over the cache maximal size, the oldest ones are discarded. If an object is modified, it is removed from the cache and becomes a "strong reference".

Caches are totally configurable in size and cleaning percent. Setting a size '0' to a cache totally disable it. By default, the high level object cache is disabled.

8.2.1 Page cache

Jalisto uses a cache mechanism for his pages. At the end of the transaction, after all modifications have been committed, there are two available behaviors:
The first one, the default behavior, consists in keeping information in memory. So we put back in cache the old modified pages, and keep the read pages.
The second one consists in dropping all pages.

User can choose the preferred behavior with the configuration option “keepInMemory”, selecting the value “yes” if he wants to keep the data between transaction boundaries, or “no” if he wants to drop them.

User can also choose the page cache size with the option “pageCacheSize”. Default value is “20”. Jalisto uses internal system page, so the page cache will not only contain instance pages.

8.2.2 New cache implementation

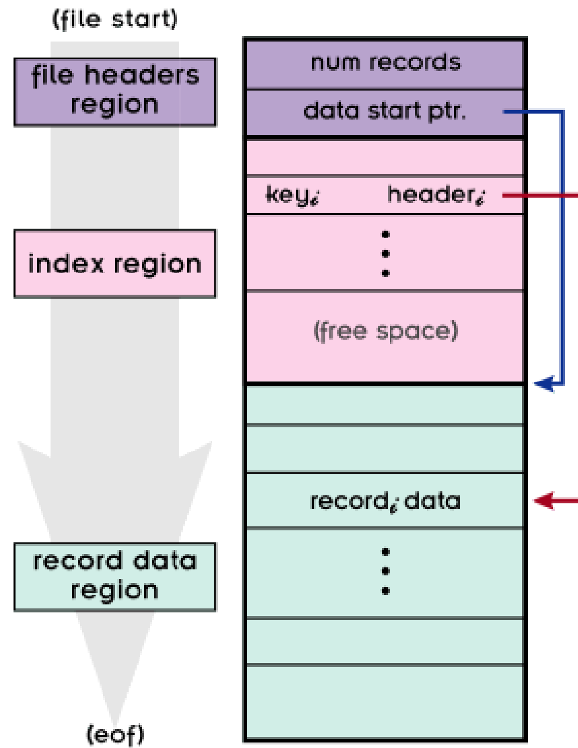
A new cache implementation can be plug in Jalisto system. The new cache must implement the JalistoCache API. The implementation cache is specified with the properties file.

8.2.3 Cache options

Key	Description	Value type	Default value
keepInMemory	Specifies if the database must keep read pages in memory between two transactions.	“yes” or “no”	“yes”
cacheClass	JalistoCache implementation class for all caches.	String	See above
objectCacheSize	Transactional object instance cache, in object count.	int	“0”
pageCacheSize	Transactional clean page cache size, in page count.	int	“20”
cacheClearPourcent	Clean percent for a full cache.	Int, from 0 to 100	“20”

8.3 Customized RandomAccessFile architecture

The file consists of three regions, each with its own format.



The records file format

8.3.1 The file headers region

This first region holds the two essential headers needed to access records in our file. The first header, called the *data start pointer,* is a `long` that points to the start of the record data. This value tells us the size of the index region. The second header, called the *num records header,* is an `int` that gives the number of records in the database. The headers region starts on the first byte of the file and extends for `FILE_HEADERS_REGION_LENGTH` bytes. We'll use `readLong()` and `readInt()` to read the headers, and `writeLong()` and `writeInt()` to write the headers.

8.3.2 The index region

Each entry in the index consists of a key and a record header. The index starts on the first byte after the file headers region and extends until the byte before the data start pointer. From this information, we can calculate a file pointer to the start of any of the n entries in the index. Entries have a fixed length -- the key data starts on the first byte in the index entry and extends `MAX_KEY_LENGTH` bytes. The corresponding record header for a given key follows immediately after the key in the index. The record header tells us where the data is located, how many bytes the record can hold, and how many bytes it is actually

holding. Index entries in the file index are in no particular order and do not map to the order in which the records are stored in the file.

8.3.3 Record data region

The record data region starts on the location indicated by the data start pointer and extends to the end of the file. Records are positioned back-to-back in the file with no free space permitted between records. This part of the file consists of raw data with no header or key information. The database file ends on the last block of the last record in the file, so there is no extra space at the end of the file. The file grows and shrinks as records are added and deleted.
The size allocated to a record does not always correspond to the actual amount of data the record contains. The record can be thought of as a container -- it may be only partially full. Valid record data is positioned at the start of the record.

9 Annex

9.1 Jalisto configuration properties summary

Key	Description	Value type	Default value
cacheClass	JalistoCache implementation class for all caches.	String	See above
cacheClearPourcent	Clean percent for a full cache.	Int, from 0 to 100	"20"
classPageSize	Number of metadatas per page.	int	"10"
concurrencyMode	If the datastore is in multi-users mode, specifies an optimistic or pessimistic concurrency mode.	"optimistic" or "pessimistic"	"optimistic"
dbFilesPaths	Path(s) of database file(s). These paths are absolute, or relative to Jalisto execution directory	Comma-separated list	<datastore name>.jalisto
dbInitialSize	Datastore initialization size, in page count.	int	"1000"
dbInitialSize	Datastore initialization size, in page count.	int	"1000"
host	Server localization.	String	"localhost"
instancePageSize	Number of instances per page.	int	"200"
internalFactoryClass'	Implementation class for the client Jalisto factory. Value MUST be set.	String	See above
keepInMemory	Specifies if the database must keep read pages in memory between two transactions.	"yes" or "no"	"yes"
logFile	Path of transaction log file. This path is absolute, or relative to Jalisto execution directory	String	"<datastore name>-log.jalisto"
name	Give a name to the datastore instance	String	"jalisto"
objectCacheSize	Transactional object instance cache, in object count.	int	"0"
oidPageSize	Number of 'logical – physical' links per page.	int	"50"
oidTableSize	Number of 'logical – physical' links in oidTable.	int	"10000"
oidTableSize	Number of 'logical – physical' links in oidTable.	int	"10000"
pageCacheSize	Transactional readed page cache size, in page count.	int	"20"
pageCacheSize	Transactional clean page cache size,	int	"20"

	in page count.		
physicalClass	The full class name of the chosen implementation of the PluggablePhysicalFileAccess interface.	String	See in chapter The storage layers
port	Communication port.	int	7777
serverPropertiesFile	Path to the server datastore configuration file.	String	See above
systemPageSize	Number of system informations per page.	int	"10"
trace	Enable logging for specified modules.	Comma-separated list	Empty list
userMode	Selection between mono-user, read-only and multi user modes.	"mono", "readonly", "multi"	"mono"