Sieben Geisslein - Userguide

Niklas Mehner

October 3, 2007

Contents

1 Connecting to the database

1.1 Local Database

A local store is opened by calling one of the openLocalStore() methods in the client class (→ org.siebengeisslein.client.Client):

```
Client.openLocalStore(String configFile);
Client openLocalStore(String configFile,
                      boolean create);
```

ConfigFile contains the path to a configuration file for the database. The first method creates a new store, if the store does not exist. The second method allows to specify wether or not to create the database.

Upon success a client (→ org.siebengeisslein.Client) instance is returned that is used for all further work on the database.

After finishing work on the database the method

```
client.close();
```

should be called.

1.1.1 Configuration File

The configuration file is a plain text file that contains mappings of the form:

key = value

Currently the following keys are defined:

dbpath Base directory for the database.

tmp Directory for temporary files (This directory contains information about transactions currently in progress. If the database crashes, this information is relevant for recovery and may not be erased).

log Directory for log files.

btree Name of the files used by the BTree implementation.

store Name of the files used by the DataStore implementation.

btreecachesize Number of BTree nodes that are cached in memory. Higher values increase performance and memory usage.

oidcachesize Size of the cache for new objects ids.

listenport Port the siebengeisslein server is listening to. If this entry is not present no listener is registered.

committhreadpoolsize Size of the thread pool that commits transactions.

messagedispatchthreads Size of the message dispatcher thread pool.

GCMaxSets Number of object changes until GC is started.

writebuffersize Size of the write buffers for transaction logs.

writebuffercount Number of writebuffers available. This should be greater than the maximum number fo concurrent transactions.

maxMsgSize Maximum size of messages received from clients. This is used to prevent clients from sending unlimited amounts of data.

flushinterval Interval to flush data to disk. Transaction logs may only be deleted after all changes have been flushed to disk. Higher values increase performance and recovery time.

httpport Port of the embedded web-server for webstart support of applications. If this entry is not present the embedded web server is not started.

callbacktimeout Timeout for callbacks to the client. If the client exeeds this time when expiring objects it is disconnected. Higher values can cause degrading server performance.

1.2 Remote Database

1.2.1 Starting a Database Server

The database is started by invoking:

```
java org.siebengeisslein.Server -c netdb.config
```

1.2.2 Connecting a Client

A remote connection is opened by invoking one of the openRemoteStore() methods in the client class.

```
Client.openRemoteStore(host, port, username, password)
```

Upon success a client (→ org.siebengeisslein.org.siebengeisslein.Client) instance is returned that is used for all further work on the database.

2 Transactions

In order to always keep the datastore in a consistent state, ACID transactions are used.

Before making any changes to the data contained in the datastore, a transaction has to be created. This is done by calling the *createTransaction()* of the client:

```
ClientTransaction ct = client.createTransaction();
```

After working on the persistent objects, the changes are committed by calling

```
 ct.commit();
```

or discarded by calling

```
 ct.abort();
```

After a transaction is closed, all persistent objects become invalid and may not be used again. Doing so may lead to erratic behavior. Despite this fact references to persistent objects can be held accross transactions. This is described in detail in section 6.

3 Optimistic Transactions

A transaction can be committed "optimistic". In that case of the ACID properties only ACI are guaranteed when the commit method returns.

```
 CommitDependency cd = ct.commitOptimistic();
```

The commit dependency can be used to track when durability is achieved:

```
cd.waitCommit()
```

Optimistic transactions allow to certain performance optimizations. They exspecially improve performance on high latency network connections and when many small transactions are used.

4 Persistent Objects

SiebenGeisslein only allows instances of classes that extend Persistent (→ org.siebengeisslein.client.Persistent) to be stored in the datastore.

These objects face some restrictions (which are verified while loading the classes):

- All fields have to be private.
- Arrays may only be used internal or passed to other objects. This means that arrays that have been assigned to fields of the object, may not be passed as arguments to methods and may not be returned from methods. Additionally arrays may not be cast to type Object.
- Only fields of the current instance ("this") may be referred in the method code.
- Fields may only be of basic types (including String), array types, interface types or of class types that extend Persistent.
- No methods may be called within constructor arguments (This is due to the java verifier rejecting legal bytecode generated by SiebenGeisslein).

The first three restrictions are needed to be able to track changes to the persistent objects. The fourth ensures, that all objects referred to by a persistent object can be made persistent.

Assigning instances of classes, that do not extend Persistent, to fields of interface type, will result in a runtime exception.

Nesting constructors will result in VerifyErrors, when the class is loaded. These errors can be removed by changing the code:

```
Object obj = new Class1(new Class2());
```

to

```
Class2 temp = new Class2();
Object obj = new Class1(temp);
```

4.1 Transient Fields

Transient fields have different semantics in SiebenGeisslein. Their content may be cleared at any time. SiebenGeisslein only guarantees, that either all or none of the transient fields of a given object are cleared.

Also if there exists a strong reference (StrongRef) to the object, the transient fields are guaranteed not to be cleared.

4.2 User Local Variables

User local variables differ from normal variable in that each user that accesses one has its own copy of the variable.

A user local variable is declared using the UserScope (→ org.siebengeisslein.client.UserScope) annotation.

```
private @UserScope PString value;
```

The type of a user local variable has to be a persistent class.

5 Root Objects

The datastore contains the transitiv hull of all root objects. All root objects are identified by their unique names. To add a new root object to the datastore the method

```
client.setRoot(name, object);
```

is called. This makes *object*, and all objects, that are referred to by *object* persistent.

An existing root object can be loaded from the datastore by calling

```
client.getRoot(name);
```

6 References

The SiebenGeisslein uses references (→ org.siebengeisslein.client.Ref) to persistent objects internally. These are invisible to users in most cases. They may however be used to keep references to persistent objects accross transaction borders. A reference to a persistent object is obtained by calling

```
Ref ref = pObject.getRef();
```

In a later transaction the persistent obejct can be loaded by calling:

```
ref.get();
```

Note that ths may result in an UnknownOIDException
(→ org.siebengeisslein.core.UnknownOIDException) if the persistent object has been garbage collected in the mean time.

7 Starting Applications

SiebenGeisslein needs a instrument persistent classes. Therefor an additional commandline argument has to be provided when starting an application.

Instead of starting the application by executing:

```
java -classpath classpath classname arguments
```

the application is now started by executing:

```
java -javaagent:instrument.jar
  -classpath SiebenGeisslein.jar:bcel.jar:classpath
  classname arguments
```

8 Hello World

This section contains the obligatory “Hello World” program. It demonstrates the basic use of the datastore. The store is used to keep track of the number of invocations of the program.

```
package org.siebengeissleinexample.hello;

import org.siebengeisslein.client.Client;
import org.siebengeisslein.client.ClientTransaction;
import org.siebengeisslein.client.Persistent;
import org.siebengeisslein.server.UnknownRootException;

public class HelloWorld extends Persistent {
    private int count;

    public HelloWorld() {
        count = 0;
    }

    public void run() {
        System.out.println("Hello World!");
        System.out.println("Called " + count + " times.");
        count++;
    }
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            throw new IllegalArgumentException("Please specify a " +
                                               "configuration file.");
        }

        // Open a connection to the datastore
        Client client = Client.openLocalStore(args[0]);

        // Create a transaction
        ClientTransaction ct = client.createTransaction();

        // Get the HelloWorld object, if it does not exist:
        // Create a new object.
        HelloWorld hw;
        try {
            hw = (HelloWorld) client.getRoot("HelloWorld");
        } catch (UnknownRootException ure) {
            hw = new HelloWorld();
            client.setRoot("HelloWorld", hw);
        }

        // Ready to run
        hw.run();

        // Commit transaction
        ct.commit();

        // Close database.
        client.close();
    }
}
```

To start the program the ApplicationStarter (→ org.siebengeisslein.client.ApplicationStarter) is used:

```
java  org.siebengeisslein.client.ApplicationStarter \
      org.siebengeissleinexample.hello.HelloWorld   \
      db.conf
```