ID Generator 3.0 Component Specification

1.Design

This component manages efficient generation of unique 64-bit IDs from an ascending sequence. Its services most closely resemble database "sequences" such as those found in Oracle. These services are implemented using a database, but do not use vendor-specific sequence generation mechanisms. Instead these services are implemented in terms of standard Java and J2EE technologies for maximum portability.

This component provides both a simple Java interface and also an EJB interface to its services.

The design presented here uses a generalized version of the "high-low" scheme used in version 1.0. The high-low scheme really reduced to allocating unique ranges of IDs and then distributing IDs from that range, and coordinating this allocation with a database table. This design simply implements this mechanism directly, and is more flexible as a result – generator instances get "blocks" of IDs from a database table, which may be configured to start on any ID and may be of any size. As of 3.0, an actual Oracle sequence-backed implementation is available, as well. This component addresses the specific requirements of 3.0 as follows:

ID Generator Plug-in Fix	IDGeneratorFactory now provides a method with the signature getIDGenerator(String idName, String implClass) that allows for specifying specific strategies. Note that backwards compatibility is maintained as the previous method is left as is (calling it is equivalent to calling getIDGenerator(idName, null)). IDGeneratorHome has also been retrofitted with a create(String implClass) method.
Oracle Sequence Plug-in	The OracleSequenceGenerator is an implementation of the IDGeneratorImpl that uses a backing Oracle DB's sequence functionality to generate IDs. A simple mapping of generator name to sequence name is provided: The string "_seq" is appended to the generator name and this is expected to be the name of the backing sequence. This can be manually overridden by calling the setSequenceName method.
Support for Large Identifiers	The IDGenerator interface now supports a getNextBigID() method that functions as described in the requirements spec
Database Access	The DB Connection Factory component is now used in lieu of the previous hardcoded DB access/lookup method

Ⓓ(For Developers) 3.0 Major Change Overview

This section summarizes the specific major changes developers will need to make and should be removed at the end of the development phase of the component.

- OracleSequenceGenerator class implementation
- Add getNextBigID method to IDGenerator interface and all implementations (including the EJB ones)
- IDGeneratorFactory.getIDGenerator(String, String) method
- IDGeneratorException now extends BaseException

- Generator implementations now use DB Connection Factory to get connections

1.1Design Patterns

➉Factory:

The IDGeneratorFactory class returns instances of implementations of IDGenerator, so that the caller is shielded from the concrete class of the implementation. IDGeneratorFactory also happens to ensure that there is at most one instance per ID sequence (per JVM).

➉Strategy:

IDGenerator is a strategy for generating IDs.

1.2Industry Standards

This component uses the following standard technologies to expose its services:

➉JDBC 3.0
➉J2EE 1.3
  ➉EJB 2.0

1.3Required Algorithms

ID generation algorithm

Pseudo-code for ID generation follows.

```
if (next ID in block <= max block ID)
        temp = next ID
        increment next ID
        return temp
else
        get block size, next sequence ID from database
        next ID = next ID from database
        max block ID = next ID + block size – 1
        return next ID
```

Oracle sequence algorithm

To generate an ID using the OracleSequenceGenerator, the default is to append "_seq" to the name of the generator to determine the name of the sequence (this can be manually overridden by clients through the setSequenceName method – use the seqName member to determine what the sequence name should be). Then, *sequenceName*.nextval will return the next value from the Oracle DB.

In order for both the IDGeneratorImpl and OracleSequenceGenerator algorithms to work the same way (high/low allocation), the Oracle generator should simulate "allocating" a block of IDs by getting the next *block size* values in a single transaction.

JDBC implementation issues

Since the database is coordinating distribution of blocks of IDs to multiple clients, potentially, it is essential that JDBC calls to the database be implemented correctly. Clients must be able to retrieve information from a database row and update it atomically.

Developers should consider the following when implementing JDBC calls:

➉Retrieving a new block of IDs requires a SELECT and UPDATE in the same transaction, therefore make sure that the Connection is **not** set to auto-commit; by default Connection do auto-commit.
➉The connection needs to use the "Repeatable Read" transaction isolation level. This ensures

that the SELECT and UPDATE are atomic, and prevents any updates in between the two. The "Serializable" isolation level is not necessary.

⓾ Make sure to set the session bean's transaction attributes appropriately in the EJB deployment descriptor – **see the example deployment descriptor in the "docs" directory.**

Here is a suggested sequence of SQL queries and updates that might be used to retrieve a block of IDs from the sequence table – developers are free to use this or not use it as desired:

```
SELECT next_block_start, block_size
FROM id_sequences
WHERE name = ?

UPDATE id_sequences
SET next_block_start = next_block_start + block_size
WHERE name = ?
```

The first query selects the next block start and current block size, which defines the next block of IDs used by the sequence. The second updates the next block start. In both cases the single in parameter is the sequence name.

1.4Component Class Overview

IDGenerator

Implementations of this interface encapsulate ID generation logic.

IDGeneratorFactory

Factory which returns instances of implementations of IDGenerator for given sequences.

IDGeneratorImpl

This class is the core of the component and actually generates the IDs. One such object exists per JVM per ID sequence. It is also used by IDGeneratorBean.

OracleSequenceGenerator

This is an alternate implementation of the IDGenerator interface that uses a backing Oracle DB with built-in sequences functionality to generate IDs.

IDGeneratorBean:

This stateless session bean is simply an EJB interface to the IDGenerator class.

IDGenerator:

IDGeneratorHome:

Remote component and home interface for the IDGeneratorBean stateless session EJB.

IDGeneratorLocal:

IDGeneratorLocalHome:

Local component and home interfaces for the IDGeneratorBean stateless session EJB.

1.5Component Exception Definitions

IDGenerationException:

Thrown whenever this component cannot retrieve ID sequence configuration or generate an ID (e.g. When unable to retrieve ID information from the database).

IDsExhaustedException:

Thrown whenever this component cannot generate an ID because a new block of IDs is needed, but there are not enough IDs left in the sequence to allocate another block.

NoSuchIDSequenceException:

Thrown whenever a non-existent ID sequence is requested (or there is no appropriate sequence defined in the Oracle DB if using the OracleSequenceGenerator).

1.6Component Benchmark and Stress Tests

ID generation through IDGenerator.getNextID() should typically be very fast – as fast as a synchronized method call. Periodically the generator needs to access the database, so performance should reflect this. That is, if the sequence block size is 20, then one should see 20 very fast ID generations, with the next one somewhat slower – time consistent with the time needed to access a database.

The ID generation mechanism is synchronized, so multiple threads should have about the same ID generation throughput as a single thread.

The IDGeneratorFactory's getIDGenerator(String, String) method should be synchronized to avoid concurrency issues there, since it modifies the generators Map. OracleSequenceGenerator's seqName data member should be volatile so that setSequenceName changes propagate to all threads.

ID generation through the EJB interface should have similar performance characteristics, but with the added overhead of the EJB call. The actual overhead is quite container-dependent. The overhead should be higher for access via the remote component interface than the local component interface.

2.Environment Requirements

2.1Environment

- J2SE 1.4 and a J2EE 1.3 classes are required to compile this component. J2SE 1.4 is required for the javax.sql classes. However, these are also provided by J2EE 1.3 distributions, so this component should also work in a J2EE 1.3 container running on J2SE 1.3.
- J2SE 1.4 and a J2EE 1.3-compliant container with EJB 2.0 support (such as Jboss) are required to test and deploy this component (see note about J2SE 1.3 above).

2.2TopCoder Software Components

- DB Connection Factory 1.0 – for getting connections to the backing DB
- Configuration Manager 2.1.3 – only because DB Connection Factory depends on it and needs to be configured via it
- Base Exception 1.0 – for wrapping exceptions

2.3Third Party Components

None.

3.Installation and Configuration

3.1Package Name

com.topcoder.util.idgenerator

3.2Configuration Parameters

None.

3.3Dependencies Configuration

To compile the component, a JAR file containing J2EE 1.3 classes (javax.ejb classes) must be present in your TopCoder "${ext_libdir}" directory. This can be downloaded from http://java.sun.com/j2ee/sdk_1.3/. Also the build.xml file must include this JAR file in the "buildlibs" path.

At runtime, this component requires a database table, defined using the following DDL fragment:

```
CREATE TABLE id_sequences (
      name               VARCHAR(255) NOT NULL,
                         PRIMARY KEY (name),
      next_block_start   BIGINT NOT NULL,
      block_size         INT NOT NULL
);
```

Note that this table stores the *next available ID* for each sequence. So, if the value of next_block_start is 10001 in some row, then 10001 has definitely not yet been assigned as an ID.

It is perfectly fine for other processes to update this table as well, as long as they use the same process as described above and update atomically.

Also, ID sequences should be configured in the database before use:

```
INSERT INTO id_sequences (name, next_block_start, block_size)
VALUES ('myGenerator', 1, 20);
```

Oracle Sequence Generation (OracleSequenceGenerator)
An appropriate sequence needs to be created within the backing Oracle database. Sequences can be created as follows:

CREATE SEQUENCE sequence_name
  MINVALUE value
  MAXVALUE value
  START WITH value
  INCREMENT BY value
  CACHE value;

The sequence will start with the “START WITH” value and generate the next value by incrementing this value with the “INCREMENT BY” value. The generated values will be within [MINVALUE, MAXVALUE]. It will cache “CACHE” IDs in memory at any one time for performance. Alternatively, “NOCACHE” may be specified instead to disable caching. Note that because of the way this component maps generator names to sequence names, the sequence_name should always end with “_seq” if possible. Otherwise, a call to OracleSequenceGenerator.setSequenceName will be required to set the correct name before use.

OracleSequenceGenerators will ignore the “next block start” entry in the id_sequences table.

Database Access
As of 3.0, the DB Connection Factory component provides database access. The DBConnectionFactory interface from that component provides a createConnection() as well as a createConnection(String name) method. Implementations of IDGenerator should access the DB as follows:

1. Call createConnection(name), passing it the IDName of the generator
2. If the above throws a DBConnectionException, call createConnection() and attempt to use it
3. If the above fails, fail with an IDGenerationException

The DB Connection Factory component should be configured appropriately via the Configuration Manager to provide named connections above. That is, if the client will be calling getIDGenerator(“myname”), there should ideally be a connection defined for DB Connection Factory called “myname”. Otherwise, the default connection will be used.

If the EJB interface will be used, then the EJB component jar file must be deployed into a J2EE

container. The specifics of EJB deployment are container-specific. **Note to developer: provide more information about deployment descriptor and deployment for common containers here.**

4.Usage Notes

4.1Required steps to test the component

- Follow the Dependencies Configuration above.
- Extract the component distribution.
- Execute “ant test” in the directory to which the distribution was extracted.

4.2Required steps to use the component

Import component classes and then simply call the getNextID() method:

```
// Simple usage of Java API
import com.topcoder.util.idgenerator.*;
...
IDGenerator myGenerator =
IDGeneratorFactory.getIDGenerator(“myGenerator”);
long nextID = myGenerator.getNextID();

// Ask for a specific implementation
myGenerator = IDGeneratorFactory.getIDGenerator(“gen”,
“com.topcoder.idgenerator.OracleSequenceGenerator”);
nextID = myGenerator.getNextID();

// Get a BigInteger ID
BigInteger nextBigID;
NextBigID = myGenerator.getNextBigID();

// Use of EJB remote interface
import com.topcoder.util.idgenerator.ejb.*;
...
Context ctx = new InitialContext();
// note: home interface JNDI name may differ depending on deployment
Object rawMyGeneratorHome =
      ctx.lookup(“java:comp/ejb/IDGeneratorHome”);
IDGeneratorHome myGeneratorHome =
      (IDGeneratorHome) PortableRemoteObject.narrow(rawMyGeneratorHome,
IDGeneratorHome.class);

IDGenerator myGenerator = myGeneratorHome.create();
long nextID = myGenerator.getNextID(“myGenerator”);


// Use of EJB local interface
import com.topcoder.util.idgenerator.ejb.*;
...
Context ctx = new InitialContext();
// note: home interface JNDI name may differ depending on deployment
IDGeneratorHome myGeneratorHome =
      (IDGeneratorHome) ctx.lookup(“java:comp/ejb/IDGeneratorHome”);

IDGenerator myGenerator = myGeneratorHome.create();
long nextID = myGenerator.getNextID(“myGenerator”);
```

4.3Demo

1) Oracle sequence setup:

```
CREATE SEQUENCE myid_seq
   MINVALUE 10
   MAXVALUE 1000
   START WITH  10
   INCREMENT BY 1
   CACHE 4;
```

2) DB Connection Factory setup (see DB Connection Factory CS for details)

3) Use

```
IDGenerator gen;
gen = IDGeneratorFactory.getIDGenerator("myid",
"com.topcoder.idgenerator.OracleSequenceGenerator");

// Default sequence name will work in this case; if it didn't we could do
// ((OracleSequenceGenerator) gen).setSequenceName("name_of_seq");

for (int i = 0; i < 5; ++i)
  System.out.println(gen.getNextID());

for (int i = 0; i < 5; ++i)
  System.out.println(gen.getNextBigID());
```

5.Future Enhancements

Future enhancements to consider include a user-definable maximum ID value, a configurable step value between IDs (for example, generating 3, 6, 9, ...), or even descending ID sequences (*Note all this functionality does currently exist if using the Oracle Sequence Generator)*. Future versions could also provide ID administration functionality, like defining new IDs, or modifying existing ones (currently this component assumes that IDs will be administered directly in the database as needed).