GUID Generator v1.0 Component Specification

1. Design

A Universally Unique Identifier (UUID) or GUID (Globally Unique Identifier) is a unique identifier can be generated without a central authority and is unique across all servers. Since a central authority is not needed, the identifiers will be generated without using any persistent data store such as a file or database. A 128 bit UUID is guaranteed to be unique for over a thousand years.

This design provides three types of generators. A 32-bit UUID generator that uses the algorithm described at http://www.mail-archive.com/ejb-interest@java.sun.com/msg01757.html and two 128-bit UUID generators that use the version 1 and version 4 implementations as described in http://www1.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt. An interface has been defined to allow the generators to be used in a consistent manner by the application.

Additionally, the UUID that is returned by the generator(s) will also implement an interface to allow the UUID to be used in a consistent manner regardless of the size. Additionally, the UUIDs can be convert to a string and then parsed back to a UUID to allow the UUID be saved to external sources.

An adapter class has been provided that will allow this component to plug into the IDGenerator component to provide 32-bit precision UUIDs. As mentioned in the future section below, creating a 64-bit precision UUID generator would be more appropriate for this adapter. The adapter uses the UUIDFactory and can easily be changed to accommodate the longer UUID.

1.1 Design Patterns

- Strategy Pattern was used in defining a set of interchangeable generators and UUID implementations
- Factory Pattern was used the UUID utility to generate the basic generators
- Adapter Pattern was used to adapt the IDGenerator's interface to using this component.

1.2 Industry Standards

UUID Draft proposal:
http://www1.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt

1.3 Required Algorithms

There are three algorithm implementations that need to be explained

1.3.1 Int32Generator

This generate creates 32-bit UUIDs based on the algorithm discussed from http://www.mail-archive.com/ejb-interest@java.sun.com/msg01757.html.

This algorithm will set itself up by:

1. Initially creating a sequence number from a random number

This algorithm then does the following to generate a number:

1. Generate a long based on the current time
2. Split the current time long into two integers
3. Add one to the sequence number
4. Create a CRC32 class
5. Update the CRC32 class with each of the integers
6. Generate the number based on the CRC32 getValue()

1.3.2 UUID Version 1 generator

This generate creates 128-bit UUIDs based on the version 1 algorithm discussed in the UUID draft proposal: http://www1.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt. This algorithm is otherwise known as the UUID time based algorithm because it's based on a time difference.

We have made certain changes to the specified algorithm to make it compatible with java. Since the MAC address is unavailable (without special native code), we substitute the IP address instead. Since the resolution of the System.currentTimeMillis() is platform independent, we check to see if we are generating two or more ID's within the same time period – if so, we increment the clock sequence to provide uniqueness. The algorithm specifies checking of the MAC address to see if it has changed from a prior state – since this component does not save any state information between restarts (as assumed by the algorithm), we ignore this part of the algorithm.

Please note that the version and variant will reflect those found in the UUID draft document stated above.

This algorithm will set itself up by:

1. Save the current time to a variable
2. Initially retrieve the IPAddress and save it for reference later
3. Generate a random number and save it to the current clock cycle.

This algorithm then does the following to generate a number:

1. If the current time is < last time (system clock was reset), regenerate a new clock cycle from a random number.
2. If the current time is = to the last time (generating numbers quicker than the resolution of System.currentTimeMillis()), add one to the clock sequence number.
3. Store the current time to the last time
4. Generate 16 byte number setting the bits in this pattern:

The most significant long can be decomposed into the following unsigned fields:

0xFFFFFFFF00000000 time_low
0x00000000FFFF0000 time_mid
0x000000000000F000 version
0x0000000000000FFF time_hi

The least significant long can be decomposed into the following unsigned fields:

0xC000000000000000 variant
0x3FFF000000000000 clock_seq
0x0000FFFFFFFFFFFF node

The version is a 4-bit number representing '0001' bits. The variant is a 2-bit number representing '10' bits.

1.3.3 UUID Version 4 generator

This generate creates 128-bit UUIDs based on the version 4 algorithm discussed in the UUID draft proposal: http://www1.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt. This algorithm is otherwise known as the UUID random based algorithm because it's based on a fully random number.

Please note that the version and variant will reflect those found in the UUID draft document stated above.

This algorithm essentially does:

1. Create a 32 byte array
2. Generate random bytes into the array by calling getRandom().nextBytes()
3. Set bit 6 to 0
4. Set bit 7 to 1
5. Set bits 12 to 15 to 0010

1.4 Component Class Overview

UUIDUtility:

This is the main class for the GUID generator component. This utility allows the application to either retrieve a prebuilt generator by its specific type or to simply generate the next UUID for that given type. This utility offers three types of generators:

- A 128 bit generator based off of the version 1draft for UUID generation
- A 128 bit generator based off of the version 4 draft for UUID generation
- A 32-bit generator based off of a published implementation.

UUIDType

This is an enum for the valid UUID types this component deals with. While java doesn't support the enum type, we can get a close resemblance by using the type safe enum component. This enum simply defines the types for the given generators in this component. The enum provides implementations of the hashCode() and equals() functions to allow it to be the key of the generators map

Generator

Defines the contract that Generators must obey. A generator will need to implement a getNextUUID() method using whatever algorithm it defines.

AbstractGenerator (implements Generator):

This abstract implementation of the Generator interface simply provides helper storage and functions to allow the generator's to be built easier. Currently, the abstract class only provides creation, storage and generation of a random generator.

Int32Generator (inherits AbstractGenerator)

This implementation of the Generator interface will generate 32-bit UUID's based on the algorithm found at http://www.mail-archive.com/ejb-interest@java.sun.com/msg01757.html.

UUIDVersion1Generator (inherits AbstractGenerator):

Creates a UUID draft version 1 generator that generates UUID based off of time as described in the UUID draft. This class will, by default, be created using the SecureRandom implementation. Application can choose to supply an application specified random implementation instead.

UUIDVersion4Generator (inherits AbstractGenerator):

Creates a UUID draft version 4 generator that generates UUID based fully off a strong random number generator. This class will, by default, be created

using the SecureRandom implementation. Application can choose to supply an application specified random implementation instead.

UUID:

Defines the contract that UUID implementations must follow. This contract allows the application to determine the number of bits the UUID represents, to return those bits and to return a string encoded version of those bits.

Abstract UUID (implements UUID):

This abstract class should be used as the base class for all UUID generators. This class provides storage of the bytes and provides default implementations of two of the UUID interface methods

UUID32Implementation (inherits AbstractUUID):

This class represents the 32-bit UUID implementation of the UUID interface. This UUID implementation will be generated from the Int32Generator.

UUID128Implementation (inherits AbstractUUID):

This class represents the 128-bit UUID implementation of the UUID interface. This UUID implementation will be generated from the UUIDVersion1 and UUIDVersion4 generators.

IDGeneratorAdapter (implements IDGenerator)

This is an adapter class to adapt the IDGenerator interface to the UUIDGenerator component. This adapter will generate id's based on the Int32Generator type.

1.5 Component Exception Definitions

There are no special exceptions defined in this design.

1.6 Thread Safety

This component deals with multiple threads and must be thread safe. This component deals with thread safety in two ways: most classes are immutable and others synchronize on a specific method. The Int32Generator and the UUIDVersion1Generator both synchronize the getNextUUID() method because they deal with state information that must be changed on each call (in addition to the UUID draft specifying a global lock being obtained). The UUIDVersion4Generator does not need synchronization lock since it generates numbers solely off a random number.

Please view the class documentation tab for each class – it will describe in detail the thread safety strategy utilized by that class.

2. Environment Requirements

2.1 Environment

- At minimum, Java1.3 is required for compilation and executing test cases.
- Java 1.2 or higher can be used for Basic and Log4j logging implementations.
- Java 1.4 or higher must be used for Java 1.4 built in logging.

2.2 TopCoder Software Components

- Type Safe Enum v1.0
- IDGenerator v3.0.2

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

- None

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.generator.guid

3.2 Configuration Parameters

This component has no configuration needed.

3.3 Dependencies Configuration

This component depends on the Type Safe Enum -which has no configuration needed.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.

- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Nothing special

4.3 Demo

The following describes how the generator will probably be used and some of its options

```
// Generate a version1 128-bit UUID
UUID uuid = UUIDFactory.getNextUUID(UUIDType.TYPE1);

// Get the bytes assigned to the uuid
byte[] bytes = uuid.toByteArray();

// Print out the UUID
System.out.println("Generated a "
      + uuid.getBitCount()
      + "-bit UUID represented as "
      + uuid.toString());

// Retrieve a UUID from a datasource
String uuidstring = getUUIDString();

// Recreate the UUID
UUID uuid = AbstractUUID.parse(uuidstring);

// Get a reference to the UUID and generate a bunch
Generator generator =
    UUIDFactory.getGenerator(UUIDType.TYPE4);

// Generate a bunch and store to a datastore
for(int x=0;x<1000;x++) {
      insertUUID(generator.getNextUUID());
```

```
      }

      // Generate a 32-bit UUID
```

```
      // Generate a version1 128-bit UUID
      UUID uuid = UUIDFactory.getNextUUID(UUIDType.TYPEINT32);

      // Get the bytes assigned to the uuid
      byte[] bytes = uuid.toByteArray();

      // Print out the UUID
      System.out.println("Generated a "
            + uuid.getBitCount()
            + "-bit UUID represented as "
            + uuid.toString());
```

```

```

```
      // Use the IDGenerator adapter
      // (Assumes the IDGenerator has been configured
      // to use the adapter)
      long id =
IDGeneratorFactory.getIDGenerator(IDGeneratorAdapter.getClass().ge
tName();

      // Create version4 UUID using a custom randomizer
      Random myRandom = getOurRandom();
      Generator gen = new UUIDVersion4Generator(myRandom);

      // Use it
      UUID uuid = gen.getNextUUID();
```

5. Future Enhancements

- The most obvious enhancement would to implement other forms of UUID generators – such as a 64-bit generator (which would better fit the IDGenerator interface) and the UUID Version 3 algorithm based of an MD5 hash of a string.
- Make the UUIDFactory configurable in which generators are available at startup.
- Expand the UUID interface to provide other helpful functions (like toLong() or toInteger(), etc).