Generating Unique Random Alpha-Numeric Non-Sequenced Identifiers

Mir Mohammed Assadullah
University of Maryland University College

Author Note

The author gratefully acknowledges helpful discussions with Travis Close, Hiep Nguyen, Peter Wages, and especially Gregory Werner.

Abstract
Every once in a while we come across the problem of generating unique identifiers that are also human readable [Bartlett] [Lee]. One may solve this problem in a number of ways. This article discusses some aspects of generating such identifiers via random number generators. The presented approach may be useful if the generated identifiers must not imply sequences. They are also useful if the user wants the ability to edit the generated identifiers.

Keywords: Unique, Random, Alpha-Numeric, Non-Sequenced, Identifier Generation, Identifier Security

Introduction

Generating Unique Random Alpha-Numeric Non-Sequenced Identifiers are utilized in instances where a computer system needs to give a unique identifier to people that have significant value associated within the system as a whole. Examples of these are airline reservation numbers, temporary passwords, electronic postage, barcodes, and human resources identification numbers. Such generated identifiers are often the first line of defense in the entire security setup. They are also used to access the associated account, reservation, etc.

There are various considerations when generating such identifiers and the choices may vary based upon the specific implementation. This paper discusses these considerations and choices.

Character Set Selection

Long identifiers may be difficult for humans to communicate effectively. In order to reduce the length of the identifiers generated, letters and numbers are combined. Also because these identifiers are supposed to be human readable, it is general practice to exclude certain letters and numbers in order to avoid confusion. Following is a table of numbers and letters generally excluded due to their similarity with other numbers and letters in certain fonts.

Table 1. Similar numbers and letters

Numbers	Letters
0	O, D
1	I, L
2	Z
8	B
	U, V

The list of these twelve characters are thus 0, 1, 2, 8, B, D, I, L, O, U, V, and Z. Additionally, only one case of letters is used, generally uppercase. Thus we are left with twenty four unique characters to work with: 3, 4, 5, 6, 7, 9, A, C, E, F, G, H, J, K, M, N, P, Q, R, S, T, W, X, Y.

Identifier Length

The first problem one may have to face is to determine the appropriate length of the generated unique identifiers. Since each character can be one of the twenty four characters, one could get 24^n unique identifiers, where *n* is the number of characters used in the string. The following table gives us some idea about how the total number of unique identifiers increases as we increase their lengths.

Table 2. Number of unique identifiers for various lengths of those identifiers.

Number of Characters	*Number of Unique Identifiers*
1	24
2	576
3	13,824
4	331,776
5	7,962,624
6	191,103,976
7	4,586,471,424
8	110,075,314,176
9	2,641,807,540,224
10	63,403,380,965,376

Identifier Generation

The following JAVA code would produce random identifiers:

```
import java.util.Random;
public class UniqueIdentifierGenerator {
 private static char[] legibleCharacters={'3','4', '5', '6', '7', '9',
    'A', 'C', 'E', 'F', 'G', 'H', 'J', 'K', 'M', 'N', 'P', 'Q', 'R',
    'S', 'T', 'W', 'X', 'Y'};
  private Random random = new Random(System.currentTimeMillis());
  public String getRandomIdentifier(int length) {
        StringBuffer sb = new StringBuffer(length);
     for (int i = 0; i < length; i++) {

sb.append(legibleCharacters[random.nextInt(legibleCharacters.length -
1) );
      }
      return sb.toString();
  }
}
```

The code above uses System.currentTimeInMillis() to seed the random number generator. The System.currentTimeInMillis() call returns back number of milliseconds since January 1, 1970 and using it as a seed for a random number generator is a relatively safe mechanism. Other mechanisms discussed in literature may provide a better alternative for a particular implementation.

Checking Availability

Once a random identifier is generated, we need to see if this identifier has been previously used or not. Generally, if this identifier is previously used, there would be a record of it in our system. If it is in the database in an indexed column, the programmer might simply query the database and find out very quickly in $O(1)$ operations.

Once a random unique identifier is generated, one might ask how many times one would have to check to see if the generated number is previously used or not. In other words, if using a database, how many database accesses one would have to perform and how many random numbers one would have to generate in order get a unique random identifier.
Suppose m is the number of unique identifiers already used, n is the total number of possible unique identifiers, and P_i is the probability of getting an unused unique random identifier in the ith try or before.

If we have already used m unique identifiers out of a total of n, the probability of finding a used identifier is m/n and so the probability of finding an unused identifier would be $1 – m/n$. Thus

$$P_1 = 1 - m/n \tag{1}$$

If we determine that the random identifier generated was already used and generate another one, the probability of this newly generated identifier to be unique would be 1 – m/n. But combining with the fact that we had already failed in our first try, the probability is $(1 - m/n)(1 - P_1)$. And finally, to find the probability that one would find a usable identifier in the first two tries, we would have to add P_1 to the term mentioned. Thus

$$P_2 = P_1 + (1 - m/n)(1 - P_1) = P_1 + (1 - m/n)(m/n) \tag{2}$$

Using similar logic, P_3 can be determined as:

$$P_3 = P_2 + (1 - m/n)(1 - P_2) = P_2 + (1 - m/n)[1 - (1 - m/n) + (1 - m/n)(m/n)]$$

Or,

$$P_3 = P_2 + (1 - m/n)[1 - 1 + m/n - m/n - (m/n)^2] = P_2 + (1 - m/n)[(m/n)^2 \tag{3}$$

Thus P_i, the probability that an available identifier is found at try *i,* is:

$$P_i = (1 - m/n) + (1 - m/n)(m/n) + (1 - m/n)(m/n)^2 + \dots + (1 - m/n)[(m/n)^{i-1} \tag{4}$$

The above can also be represented as:

$$P_i = \sum_{j=0}^{i-1} \left(1 - \frac{m}{n}\right)\left(\frac{m}{n}\right)^j$$

Multiplying both sides with *m/n*, we get:

$$(m/n)\, P_i = (1 - m/n)\,(m/n) + (1 - m/n)(m/n)^2 + (1 - m/n)\,(m/n)^3 + \dots + (1 - m/n)\,(m/n)^i$$

Or,

$$\left(\frac{m}{n}\right) P_i = \sum_{j=1}^{i} \left(1 - \frac{m}{n}\right)\left(\frac{m}{n}\right)^j$$

Subtracting above from equation (4) we get:

$$P_i - (m/n)\, P_i = (1 - m/n) - (1 - m/n)\,(m/n)^i$$
$$P_i\,(1 - m/n) = (1 - m/n)\,[1 - (m/n)^i]$$

Finally we get,

$$P_i = 1 - (m/n)^i \tag{5}$$

Where *m* is the number of unique identifiers already used, *n* is the total number of possible unique identifiers.

Thus the probability of finding a unique random non-sequenced identifier depends upon the fraction of identifiers already used and approaches 1 exponentially with the number of tries. The following table shows the probabilities when half of the possible identifiers are already used up.

Table 3. Probability of finding an unused identifier for a given try.

Try	*Probability of finding an unused identifier within and including the given try*
1	0.5
2	0.75
3	0.875
4	0.9375
5	0.96875
6	0.984375
7	0.9921875
8	0.99609375
9	0.998046875
10	0.999023438

Database Access

When planning to check the existence of randomly generated identifiers, it may be of concern to access the database too frequently. It is thus suggested that one may generate a handful of random numbers and then check them all with the database in a single query. When determining the number of identifiers as ‘handful’, the aforementioned table of probabilities, or one generated to suit your needs, may be consulted. Thus when the identifier selection pool is half full, as assumed in the table above, generating five identifiers before checking them with the database would give the programmer a 0.968 probability that one of the five identifiers is unused. Likewise, generating ten would give the programmer a probability of 0.999 that one of the identifiers is unused – even if the identifier pool is half full.

The read-only query may look something like:

```
select identifier
from records
where identifier = ‘X5FFQ’ or
      identifier = ‘CC3WF’ or
      identifier = ‘PJ44Y’
```

This query would select the existing identifiers thus not unique. Any of the remaining identifiers can then be used as valid unique identifiers. One may also use the following (Oracle flavored) query to get identifiers which are available.

```
select available_identifier from (
      select ‘X5FFQ’ as available_identifier from dual union all
      select ‘CC3WF’ from dual union all
      select ‘PJ44Y’ from dual
      )
```

```
where available_identifier not in (
    select identifier
    from records
    where identifier = ‘X5FFQ’ or
        identifier = ‘CC3WF’ or
        identifier = ‘PJ44Y’
)
```

This query would return only those identifiers that are non-existent in the database.
Here the programmer would substitute the literal strings with the generated identifiers and add more where and union clauses if necessary. Putting an index on the identifier column would execute this query in $O(1)$ time.

Caching

In case of a high frequency use of such unique identifiers, one may wish to generate a batch of identifiers, check them against existing use, and keep them in memory for faster delivery. This may be more applicable in case of a generating flight reservation numbers, order numbers of a busy online store, or something similar. The system would only need to access the database when it runs out of the available batch of cached identifiers. The code above is modified below to show a sample implementation, where the private method removeUsedIdentifiers() is database implementation dependent and left out as an exercise for the reader.

```
    private static Stack<String> cachedIdentifiers = new
Stack<String>();

    public String getCachedRandomIdentifier(int length) {
        while (cachedIdentifiers.empty()){
            int batchSize = 100;
            for (int i = 0; i < batchSize; i++) {
                cachedIdentifiers.add(getRandomIdentifier( length ));
            }
            cachedIdentifiers = removeUsedIdentifiers();
        }
        return cachedIdentifiers.pop();
    }
```

This approach would greatly reduce database access.

Mutual Exclusion

In order to avoid two threads assigning the same identifier, it is essential that the method assigning these identifiers must be executed mutually exclusively [Dijkstra]. This is also a concern when the user is allowed to edit the identifiers. This can easily be achieved by using a singleton pattern for the identifier generating object and adding the keyword synchronized to the method assigning the unique random identifiers.

Check Character
Often times another character is appended to the unique identifier in order to reduce chances of mixing it up with another number. Examples of these include Universal Product Code (UPC), Vehicle Identification Number (VIN), and International Standard Book Number (ISBN). There are various algorithms to generate the check digit [Ecker][Shulz] [Belyavskaya]. The following JAVA code implements a simple scheme of multiplying the randomly generated number with its position (left-most is 1), adding them all up, and then applying the modulo operator to this sum in order to determine the character to use.

```
import java.util.Random;

public class UniqueIdentifierGenerator {
    private static char[] legibleCharacters = {'3', '4', '5', '6',
'7', '9',
        'A', 'C', 'E', 'F', 'G', 'H', 'J', 'K', 'M', 'N', 'P', 'Q',
'R',
        'S', 'T', 'W', 'X', 'Y'};
    public String getRandomIdentifier(int length) {
        Random random = new Random();
        int checkCharRunningSum = 0;
        StringBuffer sb = new StringBuffer(length);
        for (int i = 0; i < length; i++) {
            int randomInt = random.nextInt(legibleCharacters.length -
1);
            checkCharRunningSum += randomInt * (i + 1);
            sb.append( legibleCharacters[randomInt] );
        }
        // append the check character
        sb.append(legibleCharacters[checkCharRunningSum %
legibleCharacters.length]);
        return sb.toString();
    }
}
```

Of course, if check digits are part of the identifiers, the system may either not allow the user to edit the check digit, or the identifier altogether. The advantage of using this mechanism is that a client can easily perform an initial validation of the identifier, without consulting the database – thus reducing chances of human error.

Conclusion

It is fairly inexpensive and simple to generate unique random identifiers. However, one may have to plan according to one's needs the characters to be used in the identifiers, the length of the identifiers, the number of identifiers to check against the database in a query, and the check character algorithm. This paper discussed these choices and suggested some implementations enabling the reader to make more informed choices.

References

[Ecker] ECKER, A. and POCH G., 1986. **Check Character Systems**, *Computing*, Volume 37, Number 4, December 1986, Springer Wien.

[Shulz] SHULZ, RALPH-HARDO, 2001. **Check Character Systems and Anti-symmetric Mappings**, *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2001, 136-147.

[Belyavskaya] BELYAVSKAYA, G. B., IZBASH, V. I., and MULLEN, G. L., 2005. **Check Character Systems Using Quasigroups: II**, *Designs, Codes and Cryptography*, Volume 37, Number 3, December 2005, Springer Netherlands.

[Dijkstra] DIJKSTRA, E. W., 1965. **Solution of A Problem In Concurrent Programming Control**, *Communications of the ACM*, Volume 8, Issue 9 (September 1965), Association for Computing Machinery, pp 569.

[Bartlett] BARTLETT, J. and NORRIS, D. 2009. **Adoption of Technology: An Examination of Future Behavioral Intentions Towards Distance Education**. In I. Gibson et al. (Eds.), *Proceedings of Society for Information Technology & Teacher Education International Conference 2009* Chesapeake, VA, 475-482.

[Lee] LEE, A and CHANDRA, U., 2009. **Enabling Meetings for "Anywhere and Anytime"**, *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Elisa Bertino and James B. D. Joshi (Eds), Volume 10, Springer Berlin Heidelberg,