dispatch
Internet-Draft
Updates: 4122 (if approved)
Intended status: Standards Track
Expires: August 27, 2020

BGP. Peabody
February 24, 2020

UUID Format Update
draft-peabody-dispatch-new-uuid-format-00

Abstract

This document presents a new UUID format (version 6) which is suited for use as a database key.

A common case for modern applications is to create a unique identifier to be used as a primary key in a database table that is ordered by creation time, difficult to guess and has a compact text format. None of the existing UUID versions fulfill each of these requirements. This document is a proposal to update RFC4122 with a new UUID version that addresses these concerns.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 27, 2020.

Copyright Notice

Table of Contents

1. Introduction

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Background

A lot of things have changed in the time since UUIDs were originally created. Modern applications have a need to use (and many have already implemented) UUIDs as database primary keys. However some properties of the existing specification are not well suited to this task.

The motivation for using UUIDs as database keys stems primarily from the fact that applications are increasingly distributed in nature. Simplistic "auto increment" schemes with integers in sequence do not work well in a distributed system since the effort required to synchronize such numbers across a network can easily become not worth it. The fact that UUIDs can be used to create unique and reasonably short values in distributed systems without requiring synchronization makes them a good candidate for use as a database key in such environments.

However, most of the existing UUID versions have poor database index locality. Meaning new values created in succession are not close to each other in the index and thus require inserts to be performed at random locations. The negative performance effects of which on common structures used for this (B-tree and its variants) can be dramatic. Newly inserted values should be time-ordered to address this. Version 1 UUIDs are time-ordered, but have other issues (see next).

A point of convenience and simplicity of implementation is that custom sort ordering logic should not be needed to put time ordered values in sequence. It is possible to sort Version 1 UUIDs by time but it requires breaking the bytes of the UUID into various pieces to determine the order (from the timestamp). Implementations would be simplified with a sort order where the UUID can simply be treated as an opaque sequence of bytes and ordered as such. This covers the first 64 bits of the UUID.

The latter portion (the last 64 bits) are in essence used to provide uniqueness.

Privacy and network security issues arise from using a MAC address in the node field of Version 1 UUIDs. Exposed MAC addresses can be used to locate machines to attack and can reveal various information about such machines (minimally manufacturer, potentially other details).

The use of MAC addresses in UUID Version 1, and the other hashing schemes used in the various versions, points to a more basic issue: There is no known way to guarantee "universal uniqueness". In fact, uniqueness needs are application-specific. MAC addresses in the node field might be okay for some applications. Others might be okay with using cryptographically secure random numbers (possibly with increased risk of collision). Still others might already have a predefined means to determine uniqueness for the application in question, such as a server node number. In an attempt to ensure uniqueness, the existing UUID format over-specifies exactly how this uniqueness is determined. This document posits the idea that while such mechanisms as MAC address may be okay for certain applications, it should be treated as a suggestion, not a requirement for proper implementation. Many applications will work perfectly well with more narrow and simpler uniqueness mechanisms (like using an existing node ID from whatever cluster the server is already in) and that this should be allowed as long as the uniqueness properties are clearly specified in the implementation. I.e. "using this field type as a database primary key will produce UUIDs which are unique within this database cluster" should be perfectly acceptable. Some other unnecessary requirement of global/universal uniqueness should not be needed for the implementation to be considered correct.

The property of "unguessability" is also application-specific. Some applications may desire increased security by using UUIDs which are difficult to guess (this way for example rate-limiting can be used to greatly reduce the probabililty of someone correctly guessing a new identifier or at least make it harder/take longer to do so). While applications should of course be using proper security measures, and relying solely on the unguessability of an identifier for security purposes is ill-advised, it is certainly not wrong to use this property as an additional layer of security. Examples of measures used to increase unguessability would be using cryptographically secure random data in the node and/or clock sequence fields (latter 64 bits), or using such random data in the subsecond portion of the timestamp (if subsecond time ordering is less important than unguessability for the application in question). The specification should indicate that such variations are acceptable as they do not change the format in an incompatible way.

Using a UUID as a database key generally requires communicating that UUID to other applications. The database server will store the value internally. It may be referenced in a query language (e.g. SQL), and/or transmitted in some database driver protocol. Other software, often written in another language, frequently then needs to store this identifier in its own memory and potentially perform its own operations like sorting and searching with it. And such identifiers are also commonly then used in protocols like HTTP where they indicate a particular resource. Sometimes they are typed in by humans. Sometimes constraints exist on which bytes may be used (such as an HTTP URL path). In most cases, shorter is better.

For these reasons, having a compact textual format is important. The existing hex format is already in wide use, so keeping it for backward compatibility makes sense. However an encoding using a base32 alphabet would be more compact and still be case-insensitive. A base64 alphabet would be even more compact (but require case-sensitivity). This document proposes both as options. This would allow applications to use a more compact text format for the situations needing textual representation (i.e. you can just put this value in URL and it is not unnecessarily long and does not require escaping). The alphabets used for base32 and base64 encoding should be in ASCII numeric value sequence so the text forms can also be sorted correctly as raw bytes. (This is not a property of the Base32 and Base64 standards from [RFC4648], however there are several variations in use so introducing a new one here for the express purpose of correct sorting would seem to be acceptable.)

3. Summary of Changes

The following is a summary of proposed changes to the UUID specification in [RFC4122]. Each is given as a statement of the problem or limitation to which it is addressed, along with a description of the proposed change.

3.1. Version 6

A UUID version 6 is proposed. It is ordered by creation time, sorts correctly as raw bytes, does not require use of a MAC address in the node section and has options for a compact text format.

3.2. Timestamp

The timestamp value from [RFC4122] (60-bit number of 100- nanosecond intervals since 00:00:00.00, 15 October 1582) is workable but the sequence in which the bytes are encoded (the lowest bytes first) results in unnecessary additional logic to sort correctly by timestamp. Ordering by timestamp is important for the use case of UUIDs as primary keys in a database since it improves locality by grouping new records close to each other (this can have major performance implications in large tables).

The proposed change is to encode the timestamp value into the same 60 bits as in [RFC4122] but in big-endian byte ordering. This way an application can sort by timestamp by simply treating the UUID as an opaque bunch of bytes.

3.3. Clock Sequence and Node Parts

The latter 64 bits of a UUID per [RFC4122] are the clock sequence and node fields. The node field is problematic as it encourages applications to use their MAC address which may present a security problem (it is not always appropriate to reveal the network address of a machine as it could make it the target of an attack or provide information about its manufacturer or other details). A lesser concern is that it also incidentally produces UUID with the same 6 bytes at the end and are visually more difficult to distinguish when looking at them in a list.

Seeing as the entire point of these last 64 bits is to ensure uniqueness, this document proposes that the strict definitions of clock sequence and node be relaxed. Instead implementations would be permitted to fill this section with random bytes and/or include an application defined value for uniqueness (such as a node number of a machine in a cluster).

Note for discussion: Another point to consider is that there is no known way to fully guarantee that that duplicate identifiers will not be created unless some per-determined outside source of uniqueness is employed. (Such as for version 1 UUIDs the MAC address.) However, applications each have their own requirements for uniqueness. Uniqueness within a single database cluster for example is acceptable in many cases. A specification that forces all UUIDs to be globally unique when it is not needed might not be a good idea. Identifiers are only as universally unique as their input, so it might be better to just clearly state this and say that it's fine if UUIDs are only guaranteed to be unique within a specific context if it makes sense for that application.

3.4. Alternate Text Formats

The existing UUID text format is hex encoded plus four hyphens. For many applications this is unnecessarily verbose. The same information can be encoded into significantly fewer bytes using a base 64 or base 32 alphabet.

Many applications have a need to use the unique identifier of a database record in a URL (e.g. in an HTTP request either in the path or a query parameter). It can also be useful as a file name. Being able to use a UUID for this purpose without having to escape certain characters it is a useful property.

This document proposes alternate alphabets for encoding UUIDs which are convenient for use in URLs and file names, and also sort correctly when treated as raw bytes. Some applications may not have the ability (or want) to encode and decode UUIDs from text to binary and thus having the text format also sort correctly as raw bytes is useful.

The standard Base64 and Base32 specifications in [RFC4648] do not have these properties, thus different alphabets are given for each.

Situations which require understanding the encoding SHOULD specify which encoding is used. For example, a database field which uses UUID version 6 with "b64a" encoding (see below), could be specified as type "UUID6B64A", which would result in binary storage according to UUID version 6, and otherwise read and write the value to/from applications in the b64a text format shown below. Note also that the length can be easily used to positively distinguish if a value is text or binary form. A 16-byte value will necessarily be raw unencoded bytes whereas text forms will be longer.

3.4.1. Base64 Text (Variant A)

UUIDs encoded in this form use the "url-safe base64" alphabet: "A" to "Z", "a" to "z", "0" to "9" and "-" and "_", but in ASCII value sequence. No padding characters are used.

The name "b64a" (not case sensitive) can be used by implementations to refer to this encoding.

Note: It might be useful to add another variation ("b64b") with a different alphabet. Hyphen and underscore are useful in a lot of places but there might be some others that are better for specific cases.

3.4.2. Base32 Text

Base32 can be useful if case-insensitivity is required.

UUIDs encoded in this form use digits "2" through "7" followed by "A" through "Z" (same alphabet as in [RFC4648] but in ASCII value sequence). Case is not sensitive. Implementations MAY choose to output lower case letters and doing so is also correct. Implementations which parse UUIDs encoded in this way MUST be case insensitive. No padding characters are used. Unless there is a sepcific reason for an implementation to do otherwise, it SHOULD output lower case base32 characters. The motivation for this it will increase the number of situations where UUIDs encoded in base32 and then used in different environments (some of which may be case sensitive, some not) are handled correctly by default. For example file names are case sensitive on some file systems and not on others. Preferring one specific (lower) case allows these to be used interchangably with predictable results.

The name "b32a" (not case sensitive) can be used by implementations to refer to this encoding.

4. Uniquness Service

An idea for discssion is that for applications which truly require globally unique identifiers one possible solution would be for someone to maintain a service which allocates numbers by time. In essense and for example "give me a 32-bit number that will be unique for the time range of midnight to midnight tomorrow". Such a service would be relaitvely easy to create. The effort required to maintain it depends largely on how much it is used. Applications using the same endpoint for this service would be guaranteed unique UUIDs. Companies could host their own too. I'm not sure if this sort of thing would be worth the effort but it's another idea for how to

address the global uniqueness issue for applications that really need it.

5. Acknowledgements

TODO: Acknowledgements for prior work and discussion.

6. IANA Considerations

TBD

7. Security Considerations

TODO: Provide additional information on "unguessability" as needed.

8. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

Author's Address

Brad G. Peabody

Email: brad@peabody.io