ramsey/uuid

Release 4.5.0

Ben Ramsey

2022-09-15

CONTENTS

1 Contents **3**

1.1 Introduction . . . 3
1.2 Getting Started . . . 3
1.3 RFC 4122 UUIDs . . . 5
1.4 Nonstandard UUIDs . . . 20
1.5 Using In a Database . . . 23
1.6 Customization . . . 27
1.7 Testing With UUIDs . . . 33
1.8 Upgrading ramsey/uuid . . . 35
1.9 Frequently Asked Questions (FAQs) . . . 43
1.10 Reference . . . 44
1.11 Copyright . . . 62
1.12 ramsey/uuid for Enterprise . . . 63

2 Indices and Tables **65**

PHP Namespace Index **67**

Index **69**

For ramsey/uuid 4.5.0. Updated on 2022-09-15.

Support ramsey/uuid!

Your support encourages and motivates me to continue building and maintaining open source software. If you benefit from my work, consider supporting me financially.

You may support ramsey/uuid as an individual through GitHub Sponsors or as a company through the **Tidelift Subscription**. With the Tidelift Subscription, you can get commercial maintenance and assurances, while supporting my work.

Learn more about *ramsey/uuid for enterprise*!

CHAPTER

ONE

CONTENTS

1.1 Introduction

ramsey/uuid is a PHP library for generating and working with RFC 4122 version 1, 2, 3, 4, 5, 6, and 7 universally unique identifiers (UUID). ramsey/uuid also supports optional and non-standard features, such as GUIDs and other approaches for encoding/decoding UUIDs.

1.1.1 What Is a UUID?

A universally unique identifier, or UUID, is a 128-bit unsigned integer, usually represented as a hexadecimal string split into five groups with dashes. The most widely-known and used types of UUIDs are defined by RFC 4122.

A UUID, when encoded in hexadecimal string format, looks like:

```
ebb5c735-0308-4e3c-9aea-8a270aebfe15
```

The probability of duplicating a UUID is close to zero, so they are a great choice for generating unique identifiers in distributed systems.

UUIDs can also be stored in binary format, as a string of 16 bytes.

1.2 Getting Started

1.2.1 Requirements

ramsey/uuid 4.5.0 requires the following:

- PHP 8.0+
- ext-ctype or a polyfill that provides ext-ctype, such as symfony/polyfill-ctype
- ext-json

The JSON extension is normally enabled by default, but it is possible to disable it. Other required extensions include PCRE and SPL. These standard extensions cannot be disabled without patching PHP's build system and/or C sources.

ramsey/uuid recommends installing/enabling the following extensions. While not required, these extensions improve the performance of ramsey/uuid.

- ext-gmp
- ext-bcmath

1.2.2 Install With Composer

The only supported installation method for ramsey/uuid is Composer. Use the following command to add ramsey/uuid to your project dependencies:

```
composer require ramsey/uuid
```

1.2.3 Using ramsey/uuid

After installing ramsey/uuid, the quickest way to get up-and-running is to use the static generation methods.

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid4();

printf(
    "UUID: %s\nVersion: %d\n",
    $uuid->toString(),
    $uuid->getFields()->getVersion()
);
```

This will return an instance of *Ramsey\Uuid\Rfc4122\UuidV4*.

Tip:

Use the Interfaces

Feel free to use `instanceof` to check the specific instance types of UUIDs. However, when using type hints, it's best to use the interfaces.

The most lenient interface is *Ramsey\Uuid\UuidInterface*, while *Ramsey\Uuid\Rfc4122\UuidInterface* ensures the UUIDs you're using conform to the RFC 4122 standard. If you're not sure which one to use, start with the stricter *Rfc4122\UuidInterface*.

---

ramsey/uuid provides a number of helpful static methods that help you work with and generate most types of UUIDs, without any special customization of the library.

Method	Description
Uuid::uuid1()	This generates a *Version 1: Gregorian Time* UUID.
Uuid::uuid2()	This generates a *Version 2: DCE Security* UUID.
Uuid::uuid3()	This generates a *Version 3: Name-based (MD5)* UUID.
Uuid::uuid4()	This generates a *Version 4: Random* UUID.
Uuid::uuid5()	This generates a *Version 5: Name-based (SHA-1)* UUID.
Uuid::uuid6()	This generates a *Version 6: Reordered Time* UUID.
Uuid::uuid7()	This generates a *Version 7: Unix Epoch Time* UUID.
Uuid::isValid()	Checks whether a string is a valid UUID.
Uuid::fromString()	Creates a UUID instance from a string UUID.
Uuid::fromBytes()	Creates a UUID instance from a 16-byte string.
Uuid::fromInteger()	Creates a UUID instance from a string integer.
Uuid::fromDateTime()	Creates a version 1 UUID instance from a PHP DateTimeInterface.

1.3 RFC 4122 UUIDs

1.3.1 Version 1: Gregorian Time

Attention: If you need a time-based UUID, and you don't need the other features included in version 1 UUIDs, we recommend using *version 7 UUIDs*.

A version 1 UUID uses the current time, along with the MAC address (or *node*) for a network interface on the local machine. This serves two purposes:

1. You can know *when* the identifier was created.
2. You can know *where* the identifier was created.

In a distributed system, these two pieces of information can be valuable. Not only is there no need for a central authority to generate identifiers, but you can determine what nodes in your infrastructure created the UUIDs and at what time.

Tip: It is also possible to use a **randomly-generated node**, rather than a hardware address. This is useful for when you don't want to leak machine information, while still using a UUID based on time. Keep reading to find out how.

By default, ramsey/uuid will attempt to look up a MAC address for the machine it is running on, using this value as the node. If it cannot find a MAC address, it will generate a random node.

Listing 1: Generate a version 1, Gregorian time UUID

```php
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid1();

printf(
    "UUID: %s\nVersion: %d\nDate: %s\nNode: %s\n",
    $uuid->toString(),
    $uuid->getFields()->getVersion(),
    $uuid->getDateTime()->format('r'),
    $uuid->getFields()->getNode()->toString()
);
```

This will generate a version 1 UUID and print out its string representation, the time the UUID was created, and the node used to create the UUID.

It will look something like this:

```
UUID: e22e1622-5c14-11ea-b2f3-0242ac130003
Version: 1
Date: Sun, 01 Mar 2020 23:32:15 +0000
Node: 0242ac130003
```

You may provide custom values for version 1 UUIDs, including node and clock sequence.

Listing 2: Provide custom node and clock sequence to create a version 1, Gregorian time UUID

```
use Ramsey\Uuid\Provider\Node\StaticNodeProvider;
use Ramsey\Uuid\Type\Hexadecimal;
use Ramsey\Uuid\Uuid;

$nodeProvider = new StaticNodeProvider(new Hexadecimal('121212121212'));
$clockSequence = 16383;

$uuid = Uuid::uuid1($nodeProvider->getNode(), $clockSequence);
```

Tip: Version 1 UUIDs generated in ramsey/uuid are instances of UuidV1. Check out the *Ramsey\Uuid\Rfc4122\UuidV1* API documentation to learn more about what you can do with a UuidV1 instance.

Providing a Custom Node

You may override the default behavior by passing your own node value when generating a version 1 UUID.

In the *example above*, we saw how to pass a custom node and clock sequence. An interesting thing to note about the example is its use of StaticNodeProvider. Why didn't we pass in a *Hexadecimal* value, instead?

According to RFC 4122, section 4.5, node values that do not identify the host — in other words, our own custom node value — should set the unicast/multicast bit to one (1). This bit will never be set in IEEE 802 addresses obtained from network cards, so it helps to distinguish it from a hardware MAC address.

The StaticNodeProvider sets this bit for you. This is why we used it rather than providing a *Hexadecimal* value directly.

Recall from the example that the node value we set was `121212121212`, but if you take a look at this value with `$uuid->getFields()->getNode()->toString()`, it becomes:

```
131212121212
```

That's a result of this bit being set by the StaticNodeProvider.

Generating a Random Node

Instead of providing a custom node, you may also generate a random node each time you generate a version 1 UUID. The RandomNodeProvider may be used to generate a random node value, and like the StaticNodeProvider, it also sets the unicast/multicast bit for you.

Listing 3: Provide a random node value to create a version 1, Gregorian time UUID

```
use Ramsey\Uuid\Provider\Node\RandomNodeProvider;
use Ramsey\Uuid\Uuid;

$nodeProvider = new RandomNodeProvider();

$uuid = Uuid::uuid1($nodeProvider->getNode());
```

What's a Clock Sequence?

The clock sequence part of a version 1 UUID helps prevent collisions. Since this UUID is based on a timestamp and a machine node value, it is possible for collisions to occur for multiple UUIDs generated within the same microsecond on the same machine.

The clock sequence is the solution to this problem.

The clock sequence is a 14-bit number — this supports values from 0 to 16,383 — which means it should be possible to generate up to 16,384 UUIDs per microsecond with the same node value, before hitting a collision.

> **Caution:** ramsey/uuid does not use *stable storage* for clock sequence values. Instead, all clock sequences are randomly-generated. If you are generating a lot of version 1 UUIDs every microsecond, it is possible to hit collisions because of the random values. If this is the case, you should use your own mechanism for generating clock sequence values, to ensure against randomly-generated duplicates.
>
> See section 4.2 of RFC 4122, for more information.

Privacy Concerns

As discussed earlier in this section, version 1 UUIDs use a MAC address from a local hardware network interface. This means it is possible to uniquely identify the machine on which a version 1 UUID was created.

If the value provided by the timestamp of a version 1 UUID is important to you, but you do not wish to expose the interface address of any of your local machines, see *Generating a Random Node* or *Providing a Custom Node*.

If you do not need an identifier with a timestamp value embedded in it, see *Version 4: Random* to learn about random UUIDs.

1.3.2 Version 2: DCE Security

Tip: DCE Security UUIDs are so-called because they were defined as part of the "Authentication and Security Services" for the Distributed Computing Environment (DCE) in the early 1990s.

Version 2 UUIDs are not widely used. See *Problems With Version 2 UUIDs* before deciding whether to use them.

Like a *version 1 UUID*, a version 2 UUID uses the current time, along with the MAC address (or *node*) for a network interface on the local machine. Additionally, a version 2 UUID replaces the low part of the time field with a local identifier such as the user ID or group ID of the local account that created the UUID. This serves three purposes:

1. You can know *when* the identifier was created (see *Lossy Timestamps*).
2. You can know *where* the identifier was created.
3. You can know *who* created the identifier.

In a distributed system, these three pieces of information can be valuable. Not only is there no need for a central authority to generate identifiers, but you can determine what nodes in your infrastructure created the UUIDs, at what time they were created, and the account on the machine that created them.

By default, ramsey/uuid will attempt to look up a MAC address for the machine it is running on, using this value as the node. If it cannot find a MAC address, it will generate a random node.

Listing 4: Use a domain to generate a version 2, DCE Security UUID

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid2(Uuid::DCE_DOMAIN_PERSON);

printf(
    "UUID: %s\nVersion: %d\nDate: %s\nNode: %s\nDomain: %s\nID: %s\n",
    $uuid->toString(),
    $uuid->getFields()->getVersion(),
    $uuid->getDateTime()->format('r'),
    $uuid->getFields()->getNode()->toString(),
    $uuid->getLocalDomainName(),
    $uuid->getLocalIdentifier()->toString()
);
```

This will generate a version 2 UUID and print out its string representation, the time the UUID was created, and the node used to create it, as well as the name of the local domain specified and the local domain identifier (in this case, a POSIX UID, automatically obtained from the local machine).

It will look something like this:

```
UUID: 000001f5-5e9a-21ea-9e00-0242ac130003
Version: 2
Date: Thu, 05 Mar 2020 04:30:10 +0000
Node: 0242ac130003
Domain: person
ID: 501
```

Just as with version 1 UUIDs, you may provide custom values for version 2 UUIDs, including local identifier, node, and clock sequence.

Listing 5: Provide custom identifier, node, and clock sequence to create a version 2, DCE Security UUID

```
use Ramsey\Uuid\Provider\Node\StaticNodeProvider;
use Ramsey\Uuid\Type\Hexadecimal;
use Ramsey\Uuid\Type\Integer;
use Ramsey\Uuid\Uuid;

$localId = new Integer(1001);
$nodeProvider = new StaticNodeProvider(new Hexadecimal('121212121212'));
$clockSequence = 63;

$uuid = Uuid::uuid2(
    Uuid::DCE_DOMAIN_ORG,
    $localId,
    $nodeProvider->getNode(),
    $clockSequence
);
```

Tip: Version 2 UUIDs generated in ramsey/uuid are instances of UuidV2. Check out the *Ramsey\Uuid\Rfc4122\UuidV2* API documentation to learn more about what you can do with a UuidV2 instance.

Domains

The *domain* value tells what the local identifier represents.

If using the *person* or *group* domains, ramsey/uuid will attempt to look up these values from the local machine. On POSIX systems, it will use `id -u` and `id -g`, respectively. On Windows, it will use `whoami` and `wmic`.

The *org* domain is site-defined. Its intent is to identify the organization that generated the UUID, but since this can have different meanings for different companies and projects, you get to define its value.

Table 1: DCE Security Domains

Constant	Description
Uuid::DCE_DOMAIN_PERSON	The local identifier refers to a *person* (e.g., UID).
Uuid::DCE_DOMAIN_GROUP	The local identifier refers to a *group* (e.g., GID).
Uuid::DCE_DOMAIN_ORG	The local identifier refers to an *organization* (this is site-defined).

Note: According to section 5.2.1.1 of DCE 1.1: Authentication and Security Services, the domain "can potentially hold values outside the range [0, 2^8 – 1]; however, the only values currently registered are in the range [0, 2]."

As a result, ramsey/uuid supports only the *person*, *group*, and *org* domains.

Custom and Random Nodes

In the *example above*, we provided a custom node when generating a version 2 UUID. You may also generate random node values.

To learn more, see the *Providing a Custom Node* and *Generating a Random Node* sections under *Version 1: Gregorian Time*.

Clock Sequence

In a version 2 UUID, the clock sequence serves the same purpose as in a version 1 UUID. See *What's a Clock Sequence?* to learn more.

Warning: The clock sequence in a version 2 UUID is a 6-bit number. It supports values from 0 to 63. This is different from the 14-bit number used by version 1 UUIDs.

See *Limited Uniqueness* to understand how this affects version 2 UUIDs.

Problems With Version 2 UUIDs

Version 2 UUIDs can be useful for the data they contain. However, there are trade-offs in choosing to use them.

Privacy

Unless using a randomly-generated node, version 2 UUIDs use the MAC address for a local hardware interface as the node value. In addition, they use a local identifier — usually an account or group ID. Some may consider the use of these identifying features a breach of privacy. The use of a timestamp further complicates the issue, since these UUIDs could be used to identify a user account on a specific machine at a specific time.

If you don't need an identifier with a local identifier and timestamp value embedded in it, see *Version 4: Random* to learn about random UUIDs.

Limited Uniqueness

With the inclusion of the local identifier and domain comes a serious limitation in the number of unique UUIDs that may be created. This is because:

1. The local identifier replaces the lower 32 bits of the timestamp.
2. The domain replaces the lower 8 bits of the clock sequence.

As a result, the timestamp advances — the clock *ticks* — only once every 429.49 seconds (about 7 minutes). This means the clock sequence is important to ensure uniqueness, but since the clock sequence is only 6 bits, compared to 14 bits for version 1 UUIDs, **only 64 unique UUIDs per combination of node, domain, and identifier may be generated per 7-minute tick of the clock**.

You can overcome this lack of uniqueness by using a *random node*, which provides 47 bits of randomness to the UUID — after setting the unicast/multicast bit (see discussion on *Providing a Custom Node*) — increasing the number of UUIDs per 7-minute clock tick to 2^{53} (or 9,007,199,254,740,992), at the expense of remaining locally unique.

Note: This lack of uniqueness did not present a problem for DCE, since:

> [T]he security architecture of DCE depends upon the uniqueness of security-version UUIDs *only within the context of a cell*; that is, only within the context of the local [Registration Service's] (persistent) datastore, and that degree of uniqueness can be guaranteed by the RS itself (namely, the RS maintains state in its datastore, in the sense that it can always check that every UUID it maintains is different from all other UUIDs it maintains). In other words, while security-version UUIDs are (like all UUIDs) specified to be "globally unique in space and time", security is not compromised if they are merely "locally unique per cell".
>
> —DCE 1.1: Authentication and Security Services, section 5.2.1.1

Lossy Timestamps

Version 2 UUIDs are generated in the same way as version 1 UUIDs, but the low part of the timestamp (the `time_low` field) is replaced by a 32-bit integer that represents a local identifier. Because of this, not only do version 2 UUIDs have *limited uniqueness*, but they also lack time precision.

When reconstructing the timestamp to return a DateTimeInterface instance from *`UuidV2::getDateTime()`*, we replace the 32 lower bits of the timestamp with zeros, since the local identifier should not be part of the timestamp. This results in a loss of precision, causing the timestamp to be off by a range of 0 to 429.4967295 seconds (or 7 minutes, 9 seconds, and 496,730 microseconds).

When using version 2 UUIDs, treat the timestamp as an approximation. At worst, it could be off by about 7 minutes.

Hint: If the value 429.4967295 looks familiar, it's because it directly corresponds to $2^{32} - 1$, or `0xffffffff`. The local identifier is 32-bits, and we have set each of these bits to 0, so the maximum range of timestamp drift is `0x00000000` to `0xffffffff` (counted in 100-nanosecond intervals).

1.3.3 Version 3: Name-based (MD5)

Attention: RFC 4122 states, "If backward compatibility is not an issue, SHA-1 is preferred." As a result, the use of *version 5 UUIDs* is preferred over version 3 UUIDs, unless you have a specific use-case for version 3 UUIDs.

Note: To learn about name-based UUIDs, read the section *Version 5: Name-based (SHA-1)*. Version 3 UUIDs behave exactly the same as *version 5 UUIDs*. The only difference is the hashing algorithm used to generate the UUID.

Version 3 UUIDs use MD5 as the hashing algorithm for combining the namespace and the name.

Due to the use of a different hashing algorithm, version 3 UUIDs generated with any given namespace and name will differ from version 5 UUIDs generated using the same namespace and name.

As an example, let's take a look at generating a version 3 UUID using the same namespace and name used in "*Generate a version 5, name-based UUID for a URL*."

Listing 6: Generate a version 3, name-based UUID for a URL

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid3(Uuid::NAMESPACE_URL, 'https://www.php.net');
```

Even though the namespace and name are the same, the version 3 UUID generated will always be `3f703955-aaba-3e70-a3cb-baff6aa3b28f`.

Likewise, we can use the custom namespace we created in "*Generate a custom namespace UUID*" to generate a version 3 UUID, but the result will be different from the version 5 UUID with the same custom namespace and name.

Listing 7: Use a custom namespace to create version 3, name-based UUIDs

```
use Ramsey\Uuid\Uuid;

const WIDGET_NAMESPACE = '4bdbe8ec-5cb5-11ea-bc55-0242ac130003';

$uuid = Uuid::uuid3(WIDGET_NAMESPACE, 'widget/1234567890');
```

With this custom namespace, the version 3 UUID for the name "widget/1234567890" will always be `53564aa3-4154-3ca5-ac90-dba59dc7d3cb`.

Tip: Version 3 UUIDs generated in ramsey/uuid are instances of UuidV3. Check out the *`Ramsey\Uuid\Rfc4122\UuidV3`* API documentation to learn more about what you can do with a UuidV3 instance.

1.3.4 Version 4: Random

Version 4 UUIDs are perhaps the most popular form of UUID. They are randomly-generated and do not contain any information about the time they are created or the machine that generated them. If you don't care about this information, then a version 4 UUID might be perfect for your needs.

Listing 8: Generate a version 4, random UUID

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid4();

printf(
    "UUID: %s\nVersion: %d\n",
    $uuid->toString(),
    $uuid->getFields()->getVersion()
);
```

This will generate a version 4 UUID and print out its string representation. It will look something like this:

```
UUID: 1ee9aa1b-6510-4105-92b9-7171bb2f3089
Version: 4
```

Tip: Version 4 UUIDs generated in ramsey/uuid are instances of UuidV4. Check out the *Ramsey\Uuid\Rfc4122\UuidV4* API documentation to learn more about what you can do with a UuidV4 instance.

1.3.5 Version 5: Name-based (SHA-1)

Danger: Since *version 3* and version 5 UUIDs essentially use a *salt* (the namespace) to hash data, it may be tempting to use them to hash passwords. **DO NOT do this under any circumstances!** You should not store any sensitive information in a version 3 or version 5 UUID, since MD5 and SHA-1 are insecure and have known attacks demonstrated against them. *Use these types of UUIDs as identifiers only.*

The first thing that comes to mind with most people think of a UUID is a *random* identifier, but name-based UUIDs aren't random at all. In fact, they're deterministic. For any given identical namespace and name, you will always generate the same UUID.

Name-based UUIDs are useful when you need an identifier that's based on something's *name* — think *identity* — and will always be the same no matter where or when it is created.

For example, let's say I want to create an identifier for a URL. I could use a *version 1* or *version 4* UUID to create an identifier for the URL, but what if I'm working with a distributed system, and I want to ensure that every client in this system can always generate the same identifier for any given URL?

This is where a name-based UUID comes in handy.

Name-based UUIDs combine a namespace with a name. This way, the UUIDs are unique to the namespace they're created in. RFC 4122 defines some *predefined namespaces*, one of which is for URLs.

Note: Version 5 UUIDs use SHA-1 as the hashing algorithm for combining the namespace and the name.

Listing 9: Generate a version 5, name-based UUID for a URL

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid5(Uuid::NAMESPACE_URL, 'https://www.php.net');
```

The UUID generated will always be the same, as long as the namespace and name are the same. The version 5 UUID for "https://www.php.net" in the URL namespace will always be `a8f6ae40-d8a7-58f0-be05-a22f94eca9ec`. See for yourself. Run the code above, and you'll see it always generates the same UUID.

Tip: Version 5 UUIDs generated in ramsey/uuid are instances of UuidV5. Check out the *Ramsey\Uuid\Rfc4122\UuidV5* API documentation to learn more about what you can do with a UuidV5 instance.

Custom Namespaces

If you're working with name-based UUIDs for names that don't fit into any of the *predefined namespaces*, or you don't want to use any of the predefined namespaces, you can create your own namespace.

The best way to do this is to generate a *version 1* or *version 4* UUID and save this UUID as your namespace.

Listing 10: Generate a custom namespace UUID

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid1();

printf("My namespace UUID is %s\n", $uuid->toString());
```

This will generate a version 1, Gregorian time UUID, which we'll store to a constant so we can reuse it as our own custom namespace.

Listing 11: Use a custom namespace to create version 5, name-based UUIDs

```
use Ramsey\Uuid\Uuid;

const WIDGET_NAMESPACE = '4bdbe8ec-5cb5-11ea-bc55-0242ac130003';

$uuid = Uuid::uuid5(WIDGET_NAMESPACE, 'widget/1234567890');
```

With this custom namespace, the version 5 UUID for the name "widget/1234567890" will always be `a35477ae-bfb1-5f2e-b5a4-4711594d855f`.

We can publish this namespace, allowing others to use it to generate identifiers for widgets. When two or more systems try to reference the same widget, they'll end up generating the same identifier for it, which is exactly what we want.

1.3.6 Version 6: Reordered Time

Note: Version 6, reordered time UUIDs are a new format of UUID, proposed in an Internet-Draft under review at the IETF. While the draft is still going through the IETF process, the version 6 format is not expected to change in any way that breaks compatibility.

Attention: If you need a time-based UUID, and you don't need the other features included in version 6 UUIDs, we recommend using *version 7 UUIDs*.

Version 6 UUIDs solve two problems that have long existed with the use of *version 1* UUIDs:

1. Scattered database records
2. Inability to sort by an identifier in a meaningful way (i.e., insert order)

To overcome these issues, we need the ability to generate UUIDs that are *monotonically increasing* while still providing all the benefits of version 1 UUIDs.

Version 6 UUIDs do this by storing the time in standard byte order, instead of breaking it up and rearranging the time bytes, according to the RFC 4122 definition. All other fields remain the same, and the version maintains its position, according to RFC 4122.

In all other ways, version 6 UUIDs function like version 1 UUIDs.

Tip: Prior to version 4.0.0, ramsey/uuid provided a solution for this with the *ordered-time codec*. Use of the ordered-time codec is still valid and acceptable. However, you may replace UUIDs generated using the ordered-time codec with version 6 UUIDs. Keep reading to find out how.

Listing 12: Generate a version 6, reordered time UUID

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid6();

printf(
    "UUID: %s\nVersion: %d\nDate: %s\nNode: %s\n",
```

(continues on next page)

(continued from previous page)

```
    $uuid->toString(),
    $uuid->getFields()->getVersion(),
    $uuid->getDateTime()->format('r'),
    $uuid->getFields()->getNode()->toString()
);
```

This will generate a version 6 UUID and print out its string representation, the time the UUID was created, and the node used to create the UUID.

It will look something like this:

```
UUID: 1ea60f56-b67b-61fc-829a-0242ac130003
Version: 6
Date: Sun, 08 Mar 2020 04:29:37 +0000
Node: 0242ac130003
```

You may provide custom values for version 6 UUIDs, including node and clock sequence.

Listing 13: Provide custom node and clock sequence to create a version 6, reordered time UUID

```
use Ramsey\Uuid\Provider\Node\StaticNodeProvider;
use Ramsey\Uuid\Type\Hexadecimal;
use Ramsey\Uuid\Uuid;

$nodeProvider = new StaticNodeProvider(new Hexadecimal('121212121212'));
$clockSequence = 16383;

$uuid = Uuid::uuid6($nodeProvider->getNode(), $clockSequence);
```

Tip: Version 6 UUIDs generated in ramsey/uuid are instances of UuidV6. Check out the *Ramsey\Uuid\Rfc4122\UuidV6* API documentation to learn more about what you can do with a UuidV6 instance.

Custom and Random Nodes

In the *example above*, we provided a custom node when generating a version 6 UUID. You may also generate random node values.

To learn more, see the *Providing a Custom Node* and *Generating a Random Node* sections under *Version 1: Gregorian Time*.

Clock Sequence

In a version 6 UUID, the clock sequence serves the same purpose as in a version 1 UUID. See *What's a Clock Sequence?* to learn more.

Version 1-to-6 Conversion

It is possible to convert back-and-forth between version 6 and version 1 UUIDs.

Listing 14: Convert a version 1 UUID to a version 6 UUID

```
use Ramsey\Uuid\Rfc4122\UuidV1;
use Ramsey\Uuid\Rfc4122\UuidV6;
use Ramsey\Uuid\Uuid;

$uuid1 = Uuid::fromString('3960c5d8-60f8-11ea-bc55-0242ac130003');

if ($uuid1 instanceof UuidV1) {
    $uuid6 = UuidV6::fromUuidV1($uuid1);
}
```

Listing 15: Convert a version 6 UUID to a version 1 UUID

```
use Ramsey\Uuid\Rfc4122\UuidV6;
use Ramsey\Uuid\Uuid;

$uuid6 = Uuid::fromString('1ea60f83-960c-65d8-bc55-0242ac130003');

if ($uuid6 instanceof UuidV6) {
    $uuid1 = $uuid6->toUuidV1();
}
```

Ordered-time to Version 6 Conversion

You may convert UUIDs previously generated and stored using the *ordered-time codec* into version 6 UUIDs.

Caution: If you perform this conversion, the bytes and string representation of your UUIDs will change. This will break any software that expects your identifiers to be fixed.

Listing 16: Convert an ordered-time codec encoded UUID to a version 6 UUID

```
use Ramsey\Uuid\Codec\OrderedTimeCodec;
use Ramsey\Uuid\Rfc4122\UuidV1;
use Ramsey\Uuid\Rfc4122\UuidV6;
use Ramsey\Uuid\UuidFactory;

// The bytes of a version 1 UUID previously stored in some datastore
// after encoding to bytes with the OrderedTimeCodec.
$bytes = hex2bin('11ea60faf17c8af6ad23acde48001122');

$factory = new UuidFactory();
$codec = new OrderedTimeCodec($factory->getUuidBuilder());

$factory->setCodec($codec);

$orderedTimeUuid = $factory->fromBytes($bytes);

if ($orderedTimeUuid instanceof UuidV1) {
```

(continues on next page)

(continued from previous page)

```
    $uuid6 = UuidV6::fromUuidV1($orderedTimeUuid);
}
```

Privacy Concerns

Like *version 1 UUIDs*, version 6 UUIDs use a MAC address from a local hardware network interface. This means it is possible to uniquely identify the machine on which a version 6 UUID was created.

If the value provided by the timestamp of a version 6 UUID is important to you, but you do not wish to expose the interface address of any of your local machines, see *Custom and Random Nodes*.

If you do not need an identifier with a node value embedded in it, but you still need the benefit of a monotonically increasing unique identifier, see *Version 7: Unix Epoch Time*.

1.3.7 Version 7: Unix Epoch Time

Note: Version 7, Unix Epoch time UUIDs are a new format of UUID, proposed in an Internet-Draft under review at the IETF. While the draft is still going through the IETF process, the version 7 format is not expected to change in any way that breaks compatibility.

ULIDs and Version 7 UUIDs

Version 7 UUIDs are binary-compatible with ULIDs (universally unique lexicographically-sortable identifiers).

Both use a 48-bit timestamp in milliseconds since the Unix Epoch, filling the rest with random data. Version 7 UUIDs then add the version and variant bits required by the UUID specification, which reduces the randomness from 80 bits to 74. Otherwise, they are identical.

You may even convert a version 7 UUID to a ULID. *See below for an example.*

Version 7 UUIDs solve two problems that have long existed with the use of *version 1* UUIDs:

1. Scattered database records
2. Inability to sort by an identifier in a meaningful way (i.e., insert order)

To overcome these issues, we need the ability to generate UUIDs that are *monotonically increasing*.

Version 6 UUIDs provide an excellent solution for those who need monotonically increasing, sortable UUIDs with the features of version 1 UUIDs (MAC address and clock sequence), but if those features aren't necessary for your application, using a version 6 UUID might be overkill.

Version 7 UUIDs combine random data (like version 4 UUIDs) with a timestamp (in milliseconds since the Unix Epoch, i.e., 1970-01-01 00:00:00 UTC) to create a monotonically increasing, sortable UUID that doesn't have any privacy concerns, since it doesn't include a MAC address.

For this reason, implementations should use version 7 UUIDs over versions 1 and 6, if possible.

Listing 17: Generate a version 7, Unix Epoch time UUID

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid7();
```

(continues on next page)

(continued from previous page)

```
printf(
    "UUID: %s\nVersion: %d\nDate: %s\n",
    $uuid->toString(),
    $uuid->getFields()->getVersion(),
    $uuid->getDateTime()->format('r'),
);
```

This will generate a version 7 UUID and print out its string representation and the time it was created.

It will look something like this:

```
UUID: 01833ce0-3486-7bfd-84a1-ad157cf64005
Version: 7
Date: Wed, 14 Sep 2022 16:41:10 +0000
```

To use an existing date and time to generate a version 7 UUID, you may pass a `\DateTimeInterface` instance to the `uuid7()` method.

Listing 18: Generate a version 7 UUID from an existing date and time

```
use DateTimeImmutable;
use Ramsey\Uuid\Uuid;

$dateTime = new DateTimeImmutable('@281474976710.655');
$uuid = Uuid::uuid7($dateTime);

printf(
    "UUID: %s\nVersion: %d\nDate: %s\n",
    $uuid->toString(),
    $uuid->getFields()->getVersion(),
    $uuid->getDateTime()->format('r'),
);
```

Which will print something like this:

```
UUID: ffffffff-ffff-7964-a8f6-001336ac20cb
Version: 7
Date: Tue, 02 Aug 10889 05:31:50 +0000
```

Tip: Version 7 UUIDs generated in ramsey/uuid are instances of UuidV7. Check out the *Ramsey\Uuid\Rfc4122\UuidV7* API documentation to learn more about what you can do with a UuidV7 instance.

Convert a Version 7 UUID to a ULID

As mentioned in the callout above, version 7 UUIDs are binary-compatible with ULIDs. This means you can encode a version 7 UUID using Crockford's Base 32 algorithm and it will be a valid ULID, timestamp and all.

Using the third-party library tuupola/base32, here's how we can encode a version 7 UUID as a ULID. Note that there's a little bit of work to perform the conversion, since you're working with different bases.

Listing 19: Encode a version 7, Unix Epoch time UUID as a ULID

```
use Ramsey\Uuid\Uuid;
use Tuupola\Base32;

$crockford = new Base32([
    'characters' => Base32::CROCKFORD,
    'padding' => false,
    'crockford' => true,
]);

$uuid = Uuid::uuid7();

// First, we must pad the 16-byte string to 20 bytes
// for proper conversion without data loss.
$bytes = str_pad($uuid->getBytes(), 20, "\x00", STR_PAD_LEFT);

// Use Crockford's Base 32 encoding algorithm.
$encoded = $crockford->encode($bytes);

// That 20-byte string was encoded to 32 characters to avoid loss
// of data. We must strip off the first 6 characters--which are
// all zeros--to get a valid 26-character ULID string.
$ulid = substr($encoded, 6);

printf("ULID: %s\n", $ulid);
```

This will print something like this:

```
ULID: 01GCZ05N3JFRKBRWKNGCQZGP44
```

Caution: Be aware that all version 7 UUIDs may be converted to ULIDs but not all ULIDs may be converted to UUIDs.

For that matter, all UUIDs of any version may be encoded as ULIDs, but they will not be monotonically increasing and sortable unless they are version 7 UUIDs. You will also not be able to extract a meaningful timestamp from the ULID, unless it was converted from a version 7 UUID.

RFC 4122 defines five versions of UUID, while a new Internet-Draft under review defines three new versions. Each version has different generation algorithms and properties. Which one you choose depends on your use-case. You can find out more about their applications on the specific page for that version.

Version 1: Gregorian Time This version of UUID combines a timestamp, node value (in the form of a MAC address from the local computer's network interface), and a clock sequence to ensure uniqueness. For more details, see *Version 1: Gregorian Time*.

Version 2: DCE Security This version of UUID is the same as Version 1, except the `clock_seq_low` field is replaced with a *local domain* and the `time_low` field is replaced with a *local identifier*. For more details, see *Version 2: DCE Security*.

Version 3: Name-based (MD5) This version of UUID hashes together a namespace and a name to create a deterministic UUID. The hashing algorithm used is MD5. For more details, see *Version 3: Name-based (MD5)*.

Version 4: Random This version creates a UUID using truly-random or pseudo-random numbers. For more details, see *Version 4: Random*.

Version 5: Named-based (SHA-1) This version of UUID hashes together a namespace and a name to create a deterministic UUID. The hashing algorithm used is SHA-1. For more details, see *Version 5: Name-based (SHA-1)*.

Version 6: Reordered Time This version of UUID combines the features of a *version 1 UUID* with a *monotonically increasing* UUID. For more details, see *Version 6: Reordered Time*.

Version 7: Unix Epoch Time This version of UUID combines a timestamp–based on milliseconds elapsed since the Unix Epoch–and random bytes to create a monotonically increasing, sortable UUID without the privacy and entropy concerns associated with version 1 and version 6 UUIDs. For more details, see *Version 7: Unix Epoch Time*.

1.4 Nonstandard UUIDs

1.4.1 Version 6: Reordered Time

Attention: This documentation has moved to *RFC 4122 UUIDs: Version 6: Reordered Time*.

Version 6 UUIDs have been promoted to the `Rfc4122` namespace. While still in draft form, the version 6 format is not expected to change in any way that breaks compatibility.

The *`Ramsey\Uuid\Nonstandard\UuidV6`* class is deprecated in favor of *`Ramsey\Uuid\Rfc4122\UuidV6`*.

1.4.2 Globally Unique Identifiers (GUIDs)

Tip: Using these techniques to work with GUIDs is useful if you're working with identifiers that have been stored in GUID byte order. For example, this is the case if working with the `UNIQUEIDENTIFIER` data type in Microsoft SQL Server. This is a GUID, stored as a 16-byte binary string. If working directly with the bytes, you may use the GUID functionality in ramsey/uuid to properly handle this data type.

According to the Windows Dev Center article on GUID structure, "GUIDs are the Microsoft implementation of the distributed computing environment (DCE) universally unique identifier." For all intents and purposes, a GUID string representation is identical to that of an RFC 4122 UUID. For historical reasons, *the byte order is not*.

The .NET Framework documentation explains:

> Note that the order of bytes in the returned byte array is different from the string representation of a Guid value. The order of the beginning four-byte group and the next two two-byte groups is reversed, whereas the order of the last two-byte group and the closing six-byte group is the same.

This is best explained by example.

Listing 20: Decoding a GUID from byte representation

```
use Ramsey\Uuid\FeatureSet;
use Ramsey\Uuid\UuidFactory;

// The bytes of a GUID previously stored in some datastore.
$guidBytes = hex2bin('0eab93fc9ec9584b975e9c5e68c53624');

$useGuids = true;
$featureSet = new FeatureSet($useGuids);
```

(continues on next page)

(continued from previous page)

```
$factory = new UuidFactory($featureSet);

$guid = $factory->fromBytes($guidBytes);

printf(
    "Class: %s\nGUID: %s\nVersion: %d\nBytes: %s\n",
    get_class($guid),
    $guid->toString(),
    $guid->getFields()->getVersion(),
    bin2hex($guid->getBytes())
);
```

This transforms the bytes of a GUID, as represented by `$guidBytes`, into a *`Ramsey\Uuid\Guid\Guid`* instance and prints out some details about it. It looks something like this:

```
Class: Ramsey\Uuid\Guid\Guid
GUID: fc93ab0e-c99e-4b58-975e-9c5e68c53624
Version: 4
Bytes: 0eab93fc9ec9584b975e9c5e68c53624
```

Note the difference between the string GUID and the bytes. The bytes are arranged like this:

```
0e ab 93 fc 9e c9 58 4b 97 5e 9c 5e 68 c5 36 24
```

In an RFC 4122 UUID, the bytes are stored in the same order as you see presented in the string representation. This is often called *network byte order*, or *big-endian* order. In a GUID, the order of the bytes are reversed in each grouping for the first 64 bits and stored in *little-endian* order. The remaining 64 bits are stored in network byte order. See *Endianness* to learn more.

Caution: The bytes themselves do not indicate their order. If you decode GUID bytes as a UUID or UUID bytes as a GUID, you will get the wrong values. However, you can always create a GUID or UUID from the same string value; the bytes for each will be in a different order, even though the string is the same.

The key is to know ahead of time in what order the bytes are stored. Then, you will be able to decode them using the correct approach.

Converting GUIDs to UUIDs

Continuing from the example, *Decoding a GUID from byte representation*, we can take the GUID string representation and convert it into a standard UUID.

Listing 21: Convert a GUID to a UUID

```
$uuid = Uuid::fromString($guid->toString());

printf(
    "Class: %s\nUUID: %s\nVersion: %d\nBytes: %s\n",
    get_class($uuid),
    $uuid->toString(),
    $uuid->getFields()->getVersion(),
    bin2hex($uuid->getBytes())
);
```

Because the GUID was a version 4, random UUID, this creates an instance of *Ramsey\Uuid\Rfc4122\UuidV4* from the GUID string and prints out a few details about it. It looks something like this:

```
Class: Ramsey\Uuid\Rfc4122\UuidV4
UUID: fc93ab0e-c99e-4b58-975e-9c5e68c53624
Version: 4
Bytes: fc93ab0ec99e4b58975e9c5e68c53624
```

Note how the UUID string is identical to the GUID string. However, the byte order is different, since they are in big-endian order. The bytes are now arranged like this:

```
fc 93 ab 0e c9 9e 4b 58 97 5e 9c 5e 68 c5 36 24
```

Endianness

Big-endian and little-endian refer to the ordering of bytes in a multi-byte number. Big-endian order places the most significant byte first, followed by the other bytes in descending order. Little-endian order places the least significant byte first, followed by the other bytes in ascending order.

Take the hexadecimal number `0x1234`, for example. In big-endian order, the bytes are stored as `12 34`, and in little-endian order, they are stored as `34 12`. In either case, the number is still `0x1234`.

Networking protocols usually use big-endian ordering, while computer processor architectures often use little-endian ordering. The terms originated in Jonathan Swift's *Gulliver's Travels*, where the Lilliputians argue over which end of a hard-boiled egg is the best end to crack.

1.4.3 Other Nonstandard UUIDs

Sometimes, you might encounter a string that looks like a UUID but doesn't follow the RFC 4122 specification. Take this string, for example:

```
d95959bc-2ff5-43eb-fccd-14883ba8f174
```

At a glance, this looks like a valid UUID, but the variant bits don't match RFC 4122. Instead of throwing a validation exception, ramsey/uuid will assume this is a UUID, since it fits the format and has 128 bits, but it will represent it as a *Ramsey\Uuid\Nonstandard\Uuid*.

Listing 22: Create an instance of Nonstandard\Uuid from a non-RFC 4122 UUID

```php
use Ramsey\Uuid\Uuid;

$uuid = Uuid::fromString('d95959bc-2ff5-43eb-fccd-14883ba8f174');

printf(
    "Class: %s\nUUID: %s\nVersion: %d\nVariant: %s\n",
    get_class($uuid),
    $uuid->toString(),
    $uuid->getFields()->getVersion(),
    $uuid->getFields()->getVariant()
);
```

This will create a Nonstandard\Uuid from the given string and print out a few details about it. It will look something like this:

```
Class: Ramsey\Uuid\Nonstandard\Uuid
UUID: d95959bc-2ff5-43eb-fccd-14883ba8f174
Version: 0
Variant: 7
```

Note that the version is 0. Since the variant is 7, and there is no formal specification for this variant of UUID, ramsey/uuid has no way of knowing what type of UUID this is.

Outside of RFC 4122, other types of UUIDs are in-use, following rules of their own. Some of these are on their way to becoming accepted standards, while others have historical reasons for remaining valid today. Still, others are completely random and do not follow any rules.

For these cases, ramsey/uuid provides a special functionality to handle these alternate, nonstandard forms.

Version 6: Reordered Time This is a new version of UUID that combines the features of a *version 1 UUID* with a *monotonically increasing* UUID. For more details, see *Version 6: Reordered Time*.

Globally Unique Identifiers (GUIDs) A globally unique identifier, or GUID, is often used as a synonym for UUID. A key difference is the order of the bytes. Any RFC 4122 version UUID may be represented as a GUID. For more details, see *Globally Unique Identifiers (GUIDs)*.

Other Nonstandard UUIDs Sometimes, UUID string or byte representations don't follow RFC 4122. Rather than reject these identifiers, ramsey/uuid returns them with the special Nonstandard\Uuid instance type. For more details, see *Other Nonstandard UUIDs*.

1.5 Using In a Database

Tip: ramsey/uuid-doctrine allows the use of ramsey/uuid as a Doctrine field type. If you use Doctrine, it's a great option for working with UUIDs and databases.

There are several strategies to consider when working with UUIDs in a database. Among these are whether to store the string representation or bytes and whether the UUID column should be treated as a primary key. We'll discuss a few of these approaches here, but the final decision on how to use UUIDs in a database is up to you since your needs will be different from those of others.

Note: All database code examples in this section assume the use of MariaDB and PHP Data Objects (PDO). If using a different database engine or connection library, your code will differ, but the general concepts should remain the same.

1.5.1 Storing As a String

Perhaps the easiest way to store a UUID to a database is to create a `char(36)` column and store the UUID as a string. When stored as a string, UUIDs require no special treatment in SQL statements or when displaying them.

The primary drawback is the size. At 36 characters, UUIDs can take up a lot of space, and when handling a lot of data, this can add up.

Listing 23: Create a table with a column for UUIDs

```
CREATE TABLE `notes` (
    `uuid` char(36) NOT NULL,
    `notes` text NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Using this database table, we can store the string UUID using code similar to this (assume some of the variables in this example have been set beforehand):

Listing 24: Store a string UUID to the uuid column

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid4();

$dbh = new PDO($dsn, $username, $password);

$sth = $dbh->prepare('
    INSERT INTO notes (
        uuid,
        notes
    ) VALUES (
        :uuid,
        :notes
    )
');

$sth->execute([
    ':uuid' => $uuid->toString(),
    ':notes' => $notes,
]);
```

1.5.2 Storing As Bytes

In *the previous example*, we saw how to store the string representation of a UUID to a `char(36)` column. As discussed, the primary drawback is the size. However, if we store the UUID in byte form, we only need a `char(16)` column, saving over half the space.

The primary drawback with this approach is ease-of-use. Since the UUID bytes are stored in the database, querying and selecting data becomes more difficult.

Listing 25: Create a table with a column for UUID bytes

```
CREATE TABLE `notes` (
    `uuid` char(16) NOT NULL,
    `notes` text NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Using this database table, we can store the UUID bytes using code similar to this (again, assume some of the variables in this example have been set beforehand):

Listing 26: Store UUID bytes to the uuid column

```
$sth->execute([
    ':uuid' => $uuid->getBytes(),
```

(continues on next page)

(continued from previous page)

```
    ':notes' => $notes,
]);
```

Now, when we `SELECT` the records from the database, we will need to convert the `notes.uuid` column to a ramsey/uuid object, so that we are able to use it.

Listing 27: Covert database UUID bytes to UuidInterface instance

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::uuid4();

$dbh = new PDO($dsn, $username, $password);

$sth = $dbh->prepare('SELECT uuid, notes FROM notes');
$sth->execute();

foreach ($sth->fetchAll() as $record) {
    $uuid = Uuid::fromBytes($record['uuid']);

    printf(
        "UUID: %s\nNotes: %s\n\n",
        $uuid->toString(),
        $record['notes']
    );
}
```

We'll also need to query the database using the bytes.

Listing 28: Look-up the record from the database, using the UUID bytes

```
use Ramsey\Uuid\Uuid;

$uuid = Uuid::fromString('278198d3-fa96-4833-abab-82f9e67f4712');

$dbh = new PDO($dsn, $username, $password);

$sth = $dbh->prepare('
    SELECT uuid, notes
    FROM notes
    WHERE uuid = :uuid
');

$sth->execute([
    ':uuid' => $uuid->getBytes(),
]);

$record = $sth->fetch();

if ($record) {
    $uuid = Uuid::fromBytes($record['uuid']);

    printf(
        "UUID: %s\nNotes: %s\n\n",
        $uuid->toString(),
        $record['notes']
```

(continues on next page)

(continued from previous page)

```
    );
}
```

1.5.3 Using As a Primary Key

In the previous examples, we didn't use the UUID as a primary key, but it's logical to use the `notes.uuid` field as a primary key. There's nothing wrong with this approach, but there are a couple of points to consider:

- InnoDB stores data in the primary key order
- All the secondary keys also contain the primary key (in InnoDB)

We'll deal with the first point in the section, *Insertion Order and Sorting*. For the second point, if you are using the string version of the UUID (i.e., `char(36)`), then not only will the primary key be large and take up a lot of space, but every secondary key that uses that primary key will also be much larger.

For this reason, if you choose to use UUIDs as primary keys, it might be worth the drawbacks to use UUID bytes (i.e., `char(16)`) instead of the string representation (see *Storing As Bytes*).

Hint: If not using InnoDB with MySQL or MariaDB, consult your database engine documentation to find whether it also has similar properties that will factor into your use of UUIDs.

1.5.4 Using As a Unique Key

Instead of *using UUIDs as a primary key*, you may choose to use an `AUTO_INCREMENT` column with the `int unsigned` data type as a primary key, while using a `char(36)` for UUIDs and setting a `UNIQUE KEY` on this column. This will aid in lookups while helping keep your secondary keys small.

Listing 29: Use an auto-incrementing column as primary key, with UUID as a unique key

```
CREATE TABLE `notes` (
    `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
    `uuid` char(36) NOT NULL,
    `notes` text NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `notes_uuid_uk` (`uuid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

1.5.5 Insertion Order and Sorting

UUID versions 1, 2, 3, 4, and 5 are not *monotonically increasing*. If using these versions as primary keys, the inserts will be random, and the data will be scattered on disk (for InnoDB). Over time, as the database size grows, lookups will become slower and slower.

Tip: See Percona's "Storing UUID Values in MySQL" post, for more details on the performance of UUIDs as primary keys.

To minimize these problems, two solutions have been devised:

1. *Version 6: Reordered Time* UUIDs
2. *Version 7: Unix Epoch Time* UUIDs

Note: We previously recommended the use of the *timestamp-first COMB* or *ordered-time* codecs to solve these problems. However, UUID versions 6 and 7 were defined to provide these solutions in a standardized way.

1.6 Customization

1.6.1 Ordered-time Codec

Attention: *Version 6, reordered time UUIDs* are a new version of UUID that eliminate the need for the ordered-time codec. If you aren't currently using the ordered-time codec, and you need time-based, sortable UUIDs, consider using version 6 UUIDs.

UUIDs arrange their bytes according to the standard recommended by RFC 4122. Unfortunately, this means the bytes aren't in an arrangement that supports sorting by creation time or an otherwise incrementing value. The Percona article, "Storing UUID Values in MySQL," explains at length the problems this can cause. It also recommends a solution: the *ordered-time UUID*.

RFC 4122 version 1, Gregorian time UUIDs rearrange the bytes of the time fields so that the lowest bytes appear first, the middle bytes are next, and the highest bytes come last. Logical sorting is not possible with this arrangement.

An ordered-time UUID is a version 1 UUID with the time fields arranged in logical order so that the UUIDs can be sorted by creation time. These UUIDs are *monotonically increasing*, each one coming after the previously-created one, in a proper sort order.

Listing 30: Use the ordered-time codec to generate a version 1 UUID

```
use Ramsey\Uuid\Codec\OrderedTimeCodec;
use Ramsey\Uuid\UuidFactory;

$factory = new UuidFactory();
$codec = new OrderedTimeCodec($factory->getUuidBuilder());

$factory->setCodec($codec);

$orderedTimeUuid = $factory->uuid1();

printf(
    "UUID: %s\nVersion: %d\nDate: %s\nNode: %s\nBytes: %s\n",
    $orderedTimeUuid->toString(),
    $orderedTimeUuid->getFields()->getVersion(),
    $orderedTimeUuid->getDateTime()->format('r'),
    $orderedTimeUuid->getFields()->getNode()->toString(),
    bin2hex($orderedTimeUuid->getBytes())
);
```

This will use the ordered-time codec to generate a version 1 UUID and will print out details about the UUID similar to these:

```
UUID: 593200aa-61ae-11ea-bbf2-0242ac130003
Version: 1
Date: Mon, 09 Mar 2020 02:33:23 +0000
Node: 0242ac130003
Bytes: 11ea61ae593200aabbf20242ac130003
```

Attention: Only the byte representation is rearranged. The string representation follows the format of a standard version 1 UUID. This means only the byte representation of an ordered-time codec encoded UUID may be used for sorting, such as with database results.

To store the byte representation to a database field, see *Storing As Bytes*.

Hint: If you use this codec and store the bytes of the UUID to the database, as recommended above, you will need to use this codec to decode the bytes, as well. Otherwise, the UUID string value will be incorrect.

```
// Using a factory configured with the OrderedTimeCodec, as shown above.
$orderedTimeUuid = $factory->fromBytes($bytes);
```

1.6.2 Timestamp-first COMB Codec

Attention: *Version 7, Unix Epoch time UUIDs* are a new version of UUID that eliminate the need for the timestamp-first COMB codec. If you aren't currently using the timestamp-first COMB codec, and you need time-based, sortable UUIDs, consider using version 7 UUIDs.

Version 4, random UUIDs are doubly problematic when it comes to sorting and storing to databases (see *Insertion Order and Sorting*), since their values are random, and there is no timestamp associated with them that may be rearranged, like with the *ordered-time codec*. In 2002, Jimmy Nilsson recognized this problem with random UUIDs and proposed a solution he called "COMBs" (see "The Cost of GUIDs as Primary Keys").

So-called because they *combine* random bytes with a timestamp, the timestamp-first COMB codec replaces the first 48 bits of a version 4, random UUID with a Unix timestamp and microseconds, creating an identifier that can be sorted by creation time. These UUIDs are *monotonically increasing*, each one coming after the previously-created one, in a proper sort order.

Listing 31: Use the timestamp-first COMB codec to generate a version 4 UUID

```
use Ramsey\Uuid\Codec\TimestampFirstCombCodec;
use Ramsey\Uuid\Generator\CombGenerator;
use Ramsey\Uuid\UuidFactory;

$factory = new UuidFactory();
$codec = new TimestampFirstCombCodec($factory->getUuidBuilder());

$factory->setCodec($codec);

$factory->setRandomGenerator(new CombGenerator(
    $factory->getRandomGenerator(),
    $factory->getNumberConverter()
```

(continues on next page)

(continued from previous page)

```
));

$timestampFirstComb = $factory->uuid4();

printf(
    "UUID: %s\nVersion: %d\nBytes: %s\n",
    $timestampFirstComb->toString(),
    $timestampFirstComb->getFields()->getVersion(),
    bin2hex($timestampFirstComb->getBytes())
);
```

This will use the timestamp-first COMB codec to generate a version 4 UUID with the timestamp replacing the first 48 bits and will print out details about the UUID similar to these:

```
UUID: 9009ebcc-cd99-4b5f-90cf-9155607d2de9
Version: 4
Bytes: 9009ebcccd994b5f90cf9155607d2de9
```

Note that the bytes are in the same order as the string representation. Unlike the *ordered-time codec*, the timestamp-first COMB codec affects both the string representation and the byte representation. This means either the string UUID or the bytes may be stored to a datastore and sorted. To learn more, see *Using In a Database*.

1.6.3 Using a Custom Calculator

By default, ramsey/uuid uses brick/math as its internal calculator. However, you may change the calculator, if your needs require something else.

To swap the default calculator with your custom one, first make an adapter that wraps your custom calculator and implements `Ramsey\Uuid\Math\CalculatorInterface`. This might look something like this:

Listing 32: Create a custom calculator wrapper that implements CalculatorInterface

```
namespace MyProject;

use Other\OtherCalculator;
use Ramsey\Uuid\Math\CalculatorInterface;
use Ramsey\Uuid\Type\Integer as IntegerObject;
use Ramsey\Uuid\Type\NumberInterface;

class MyUuidCalculator implements CalculatorInterface
{
    private $internalCalculator;

    public function __construct(OtherCalculator $customCalculator)
    {
        $this->internalCalculator = $customCalculator;
    }

    public function add(NumberInterface $augend, NumberInterface ...$addends):␣
↪NumberInterface
    {
        $value = $augend->toString();

        foreach ($addends as $addend) {
```

(continues on next page)

(continued from previous page)

```
            $value = $this->internalCalculator->plus($value, $addend->toString());
        }

        return new IntegerObject($value);
    }

    /* ... Class truncated for brevity ... */

}
```

The easiest way to use your custom calculator wrapper is to instantiate a new FeatureSet, set the calculator on it, and pass the FeatureSet into a new UuidFactory. Using the factory, you may then generate and work with UUIDs, using your custom calculator.

Listing 33: Use your custom calculator wrapper when working with UUIDs

```
use MyProject\MyUuidCalculator;
use Other\OtherCalculator;
use Ramsey\Uuid\FeatureSet;
use Ramsey\Uuid\UuidFactory;

$otherCalculator = new OtherCalculator();
$myUuidCalculator = new MyUuidCalculator($otherCalculator);

$featureSet = new FeatureSet();
$featureSet->setCalculator($myUuidCalculator);

$factory = new UuidFactory($featureSet);

$uuid = $factory->uuid1();
```

1.6.4 Using a Custom Validator

By default, ramsey/uuid validates UUID strings with the lenient validator *Ramsey\Uuid\Validator\GenericValidator*. This validator ensures the string is 36 characters, has the dashes in the correct places, and uses only hexadecimal values. It does not ensure the string is of the RFC 4122 variant or contains a valid version.

The validator *Ramsey\Uuid\Rfc4122\Validator* validates UUID strings to ensure they match the RFC 4122 variant and contain a valid version. Since it is not enabled by default, you will need to configure ramsey/uuid to use it, if you want stricter validation.

Listing 34: Set an alternate validator to use for Uuid::isValid()

```
use Ramsey\Uuid\Rfc4122\Validator as Rfc4122Validator;
use Ramsey\Uuid\Uuid;
use Ramsey\Uuid\UuidFactory;

$factory = new UuidFactory();
$factory->setValidator(new Rfc4122Validator());

Uuid::setFactory($factory);

if (!Uuid::isValid('2bfb5006-087b-9553-5082-e8f39337ad29')) {
```

(continues on next page)

(continued from previous page)

```
    echo "This UUID is not valid!\n";
}
```

Tip: If you want to use your own validation, create a class that implements *Ramsey\Uuid\Validator\ValidatorInterface* and use the same method to set your validator on the factory.

1.6.5 Replace the Default Factory

In many of the examples throughout this documentation, we've seen how to configure the factory and then use that factory to generate and work with UUIDs.

For example:

Listing 35: Configure the factory and use it to generate a version 1 UUID

```
use Ramsey\Uuid\Codec\OrderedTimeCodec;
use Ramsey\Uuid\UuidFactory;

$factory = new UuidFactory();
$codec = new OrderedTimeCodec($factory->getUuidBuilder());

$factory->setCodec($codec);

$orderedTimeUuid = $factory->uuid1();
```

When doing this, the default behavior of ramsey/uuid is left intact. If we call `Uuid::uuid1()` to generate a version 1 UUID after configuring the factory as shown above, it won't use *OrderedTimeCodec* to generate the UUID.

Listing 36: The behavior differs between $factory->uuid1() and Uuid::uuid1()

```
$orderedTimeUuid = $factory->uuid1();

printf(
    "UUID: %s\nBytes: %s\n\n",
    $orderedTimeUuid->toString(),
    bin2hex($orderedTimeUuid->getBytes())
);

$uuid = Uuid::uuid1();

printf(
    "UUID: %s\nBytes: %s\n\n",
    $uuid->toString(),
    bin2hex($uuid->getBytes())
);
```

In this example, we print out details for two different UUIDs. The first was generated with the *OrderedTimeCodec* using `$factory->uuid1()`. The second was generated using `Uuid::uuid1()`. It looks something like this:

```
UUID: 2ff06620-6251-11ea-9791-0242ac130003
Bytes: 11ea62512ff0662097910242ac130003
```

(continues on next page)

(continued from previous page)

```
UUID: 2ff09730-6251-11ea-ba64-0242ac130003
Bytes: 2ff09730625111eaba640242ac130003
```

Notice the arrangement of the bytes. The first set of bytes has been rearranged, according to the ordered-time codec rules, but the second set of bytes remains in the same order as the UUID string.

Configuring the factory does not change the default behavior.

If we want to change the default behavior, we must *replace* the factory used by the Uuid static methods, and we can do this using the *Uuid::setFactory()* static method.

Listing 37: Replace the factory to globally affect Uuid behavior

```
Uuid::setFactory($factory);

$uuid = Uuid::uuid1();
```

Now, every time we call *Uuid::uuid()*, ramsey/uuid will use the factory configured with the *OrderedTimeCodec* to generate version 1 UUIDs.

Warning: Calling *Uuid::setFactory()* to replace the factory will change the behavior of Uuid no matter where it is used, so keep this in mind when replacing the factory. If you replace the factory deep inside a method somewhere, any later code that calls a static method on *Ramsey\Uuid\Uuid* will use the new factory to generate UUIDs.

ramsey/uuid offers a variety of ways to modify the standard behavior of the library through dependency injection. Using FeatureSet, UuidFactory, and *Uuid::setFactory()*, you are able to replace just about any builder, codec, converter, generator, provider, and more.

Ordered-time Codec The ordered-time codec exists to rearrange the bytes of a version 1, Gregorian time UUID so that the timestamp portion of the UUID is monotonically increasing. To learn more, see *Ordered-time Codec*.

Timestamp-first COMB Codec The timestamp-first COMB codec replaces part of a version 4, random UUID with a timestamp, so that the UUID becomes monotonically increasing. To learn more, see *Timestamp-first COMB Codec*.

Using a Custom Calculator It's possible to replace the default calculator ramsey/uuid uses. If your requirements require a different solution for making calculations, see *Using a Custom Calculator*.

Using a Custom Validator If your requirements require a different level of validation or a different UUID format, you may replace the default validator. See *Using a Custom Validator*, to learn more.

Replace the Default Factory Not only are you able to inject alternate builders, codecs, etc. into the factory and use the factory to generate UUIDs, you may also replace the global, static factory used by the static methods on the Uuid class. To find out how, see *Replace the Default Factory*.

1.7 Testing With UUIDs

One problem with the use of `final` is the inability to create a mock object to use in tests. However, the following techniques should help with testing.

Tip: To learn why ramsey/uuid uses `final`, take a look at *Why does ramsey/uuid use final?*.

1.7.1 Inject a UUID of a Specific Type

Let's say we have a method that uses a type hint for *UuidV1*.

```
public function tellTime(UuidV1 $uuid): string
{
    return $uuid->getDateTime()->format('Y-m-d H:i:s');
}
```

Since this method uses UuidV1 as the type hint, we're not able to pass another object that implements UuidInterface, and we cannot extend or mock UuidV1, so how do we test this?

One way is to use *Uuid::uuid1()* to create a regular UuidV1 instance and pass it.

```
public function testTellTime(): void
{
    $uuid = Uuid::uuid1();
    $myObj = new MyClass();

    $this->assertIsString($myObj->tellTime($uuid));
}
```

This might satisfy our testing needs if we only want to assert that the method returns a string. If we want to test for a specific string, we can do that, too, by generating a UUID ahead of time and using it as a known value.

```
public function testTellTime(): void
{
    // We generated this version 1 UUID ahead of time and know the
    // exact date and time it contains, so we can use it to test the
    // return value of our method.
    $uuid = Uuid::fromString('177ef0d8-6630-11ea-b69a-0242ac130003');
    $myObj = new MyClass();

    $this->assertSame('2020-03-14 20:12:12', $myObj->tellTime($uuid));
}
```

Note: These examples assume the use of PHPUnit for tests. The concepts will work no matter what testing framework you use.

1.7.2 Returning Specific UUIDs From a Static Method

Sometimes, rather than pass UUIDs as method arguments, we might call the static methods on the Uuid class from inside the method we want to test. This can be tricky to test.

```
public function tellTime(): string
{
    $uuid = Uuid::uuid1();

    return $uuid->getDateTime()->format('Y-m-d H:i:s');
}
```

We can call this in a test and assert that it returns a string, but we can't return a specific UUID value from the static method call — or can we?

We can do this by *overriding the default factory*.

First, we create our own factory class for testing. In this example, we extend UuidFactory, but you may create your own separate factory class for testing, as long as you implement *Ramsey\Uuid\UuidFactoryInterface*.

```
namespace MyPackage;

use Ramsey\Uuid\UuidFactory;
use Ramsey\Uuid\UuidInterface;

class MyTestUuidFactory extends UuidFactory
{
    public $uuid1;

    public function uuid1($node = null, ?int $clockSeq = null): UuidInterface
    {
        return $this->uuid1;
    }
}
```

Now, from our tests, we can replace the default factory with our new factory, and we can even change the value returned by the *uuid1()* method for our tests.

```
/**
 * @runInSeparateProcess
 * @preserveGlobalState disabled
 */
public function testTellTime(): void
{
    $factory = new MyTestUuidFactory();
    Uuid::setFactory($factory);

    $myObj = new MyClass();

    $factory->uuid1 = Uuid::fromString('177ef0d8-6630-11ea-b69a-0242ac130003');
    $this->assertSame('2020-03-14 20:12:12', $myObj->tellTime());

    $factory->uuid1 = Uuid::fromString('13814000-1dd2-11b2-9669-00007ffffffe');
    $this->assertSame('1970-01-01 00:00:00', $myObj->tellTime());
}
```

Attention: The factory is a static property on the Uuid class. By replacing it like this, all uses of the Uuid class after this point will continue to use the new factory. This is why we must run the test in a separate process. Otherwise, this could cause other tests to fail.

Running tests in separate processes can significantly slow down your tests, so try to use this technique sparingly, and if possible, pass your dependencies to your objects, rather than creating (or fetching them) from within. This makes testing easier.

1.7.3 Mocking UuidInterface

Another technique for testing with UUIDs is to mock *UuidInterface*.

Consider a method that accepts a UuidInterface.

```
public function tellTime(UuidInterface $uuid): string
{
    return $uuid->getDateTime()->format('Y-m-d H:i:s');
}
```

We can mock UuidInterface, passing that mocked value into this method. Then, we can make assertions about what methods were called on the mock object. In the following example test, we don't care whether the return value matches an actual date format. What we care about is that the methods on the UuidInterface object were called.

```
public function testTellTime(): void
{
    $dateTime = Mockery::mock(DateTime::class);
    $dateTime->expects()->format('Y-m-d H:i:s')->andReturn('a test date');

    $uuid = Mockery::mock(UuidInterface::class, [
        'getDateTime' => $dateTime,
    ]);

    $myObj = new MyClass();

    $this->assertSame('a test date', $myObj->tellTime($uuid));
}
```

Note: One of my favorite mocking libraries is Mockery, so that's what I use in these examples. However, other mocking libraries exist, and PHPUnit provides built-in mocking capabilities.

1.8 Upgrading ramsey/uuid

1.8.1 Version 3 to 4

I've made great efforts to ensure that the upgrade experience for most will be seamless and uneventful. However, no matter the degree to which you use ramsey/uuid (customized or unchanged), there are a number of things to be aware of as you upgrade your code to use version 4.

Tip: These are the changes that are most likely to affect you. For a full list of changes, take a look at the 4.0.0

changelog.

What's New?

There are a lot of new features in ramsey/uuid! Here are a few of them:

- Support *version 6 UUIDs*.
- Support *version 2 (DCE Security) UUIDs*.
- Add classes to represent each version of RFC 4122 UUID. When generating new UUIDs or creating UUIDs from existing strings, bytes, or integers, if the UUID is an RFC 4122 variant, one of these instances will be returned:
  - `Rfc4122\UuidV1`
  - `Rfc4122\UuidV2`
  - `Rfc4122\UuidV3`
  - `Rfc4122\UuidV4`
  - `Rfc4122\UuidV5`
  - `Rfc4122\NilUuid`
- Add classes to represent version 6 UUIDs, GUIDs, and nonstandard (non-RFC 4122 variants) UUIDs:
  - `Nonstandard\UuidV6`
  - `Nonstandard\Uuid`
  - `Guid\Guid`
- Add `Uuid::fromDateTime()` to create version 1 UUIDs from instances of DateTimeInterface.

What's Changed?

Attention: ramsey/uuid version 4 requires PHP 7.2 or later.

Quite a bit has changed, but much remains familiar. Unless you've changed the behavior of ramsey/uuid through custom codecs, providers, generators, etc., the standard functionality and API found in version 3 will not differ much.

Here are the highlights:

- ramsey/uuid now works on 32-bit and 64-bit systems, with no degradation in functionality! All Degraded* classes are deprecated and no longer used; they'll go away in ramsey/uuid version 5.
- Pay attention to the *return types for the static methods* on the `Uuid` class. They've changed slightly, but this won't affect you if your type hints use `UuidInterface`.
- The *return types for three methods* defined on `UuidInterface` have changed, breaking backwards compatibility. **Take note and update your code.**
- *There are a number of deprecations.* These shouldn't affect you now, but please take a look at the recommendations and update your code soon. These will go away in ramsey/uuid version 5.

- ramsey/uuid now *throws custom exceptions for everything*. The exception UnsatisfiedDependencyException no longer exists.
- If you customize ramsey/uuid at all by implementing the interfaces, take a look at the *interface* and *constructor* changes and update your code.

Tip: If you maintain a public project that uses ramsey/uuid version 3 and you find that **your code does not require any changes to upgrade** to version 4, consider using the following version constraint in your project's `composer.json` file:

```
composer require ramsey/uuid:"^3 || ^4"
```

This will allow any downstream users of your project who aren't ready to upgrade to version 4 the ability to continue using your project while deciding on an appropriate upgrade schedule.

If your downstream users do not specify ramsey/uuid as a dependency, and they use functionality specific to version 3, they may need to update their own Composer dependencies to use ramsey/uuid `^3` to avoid using version 4.

Uuid Static Methods

All the static methods on the *Uuid* class continue to work as they did in version 3, with this slight change: **they now return more-specific types**, all of which implement the new interface *Rfc4122\UuidInterface*, which implements the familiar interface *UuidInterface*.

If your type hints are for *UuidInterface*, then you should not require any changes.

Table 2: Return types for Uuid static methods

Method	3.x Returned	4.x Returns
Uuid::uuid1()	*Uuid*	*Rfc4122\UuidV1*
Uuid::uuid3()	*Uuid*	*Rfc4122\UuidV3*
Uuid::uuid4()	*Uuid*	*Rfc4122\UuidV4*
Uuid::uuid5()	*Uuid*	*Rfc4122\UuidV5*

Uuid::fromString(), *Uuid::fromBytes()*, and *Uuid::fromInteger()* all return an appropriate more-specific type, based on the input value. If the input value is a version 1 UUID, for example, the return type will be an *Rfc4122\UuidV1*. If the input looks like a UUID or is a 128-bit number, but it doesn't validate as an RFC 4122 UUID, the return type will be a *Nonstandard\Uuid*. These return types implement *UuidInterface*. If using this as a type hint, you shouldn't need to make any changes.

Changed Return Types

The following *UuidInterface* method return types have changed in version 4 and you will need to update your code, if you use these methods.

Table 3: Changed UuidInterface method return types

Method	3.x Returned	4.x Returns
UuidInterface::getFields()	`array`	*Rfc4122\FieldsInterface*
UuidInterface::getHex()	`string`	*Type\Hexadecimal*
UuidInterface::getInteger()	`mixed`[1]	*Type\Integer*

In version 3, the following *Uuid* methods return `int`, `string`, or Moontoast\Math\BigNumber, depending on the environment. In version 4, they all return numeric `string` values for the sake of consistency. These methods *are also deprecated* and will be removed in version 5.

- `getClockSeqHiAndReserved()`
- `getClockSeqLow()`
- `getClockSequence()`
- `getLeastSignificantBits()`
- `getMostSignificantBits()`
- `getNode()`
- `getTimeHiAndVersion()`
- `getTimeLow()`
- `getTimeMid()`
- `getTimestamp()`

Deprecations

UuidInterface

The following *UuidInterface* methods are deprecated, but upgrading to version 4 should not cause any problems if using these methods. You are encouraged to update your code according to the recommendations, though, since these methods will go away in version 5.

Table 4: Deprecated UuidInterface methods

Deprecated Method	Update To
`getDateTime()`	Use `getDateTime()` on *UuidV1*, *UuidV2*, or *UuidV6*
`getClockSeqHiAndReservedHex()`	*getFields()->getClockSeqHiAndReserved()->toString()*
`getClockSeqLowHex()`	*getFields()->getClockSeqLow()->toString()*
`getClockSequenceHex()`	*getFields()->getClockSeq()->toString()*
`getFieldsHex()`	*getFields()*[2]
`getLeastSignificantBitsHex()`	`substr($uuid->getHex()->toString(), 0, 16)`
`getMostSignificantBitsHex()`	`substr($uuid->getHex()->toString(), 16)`
`getNodeHex()`	*getFields()->getNode()->toString()*
`getNumberConverter()`	This method has no replacement; plan accordingly.
`getTimeHiAndVersionHex()`	*getFields()->getTimeHiAndVersion()->toString()*
`getTimeLowHex()`	*getFields()->getTimeLow()->toString()*
`getTimeMidHex()`	*getFields()->getTimeMid()->toString()*
`getTimestampHex()`	*getFields()->getTimestamp()->toString()*
`getUrn()`	`Ramsey\Uuid\Rfc4122\UuidInterface::getUrn`
`getVariant()`	*getFields()->getVariant()*
`getVersion()`	*getFields()->getVersion()*

[1] This `mixed` return type could have been an `int`, `string`, or Moontoast\Math\BigNumber. In version 4, ramsey/uuid cleans this up for the sake of consistency.

[2] The *getFields()* method returns a *Type\Hexadecimal* instance; you will need to construct an array if you wish to match the return value of the deprecated `getFieldsHex()` method.

Uuid

Uuid as an instantiable class is deprecated. In ramsey/uuid version 5, its constructor will be `private`, and the class will be `final`. For more information, see *Why does ramsey/uuid use final?*

Note: *Uuid* is being replaced by more-specific concrete classes, such as:

- *Rfc4122\UuidV1*
- *Rfc4122\UuidV3*
- *Rfc4122\UuidV4*
- *Rfc4122\UuidV5*
- *Nonstandard\Uuid*

However, the *Uuid* class isn't going away. It will still hold common constants and static methods.

- `Uuid::UUID_TYPE_IDENTIFIER` is deprecated. Use `Uuid::UUID_TYPE_DCE_SECURITY` instead.
- `Uuid::VALID_PATTERN` is deprecated. Use the following instead:

```
use Ramsey\Uuid\Validator\GenericValidator;
use Ramsey\Uuid\Rfc4122\Validator as Rfc4122Validator;

$genericPattern = (new GenericValidator())->getPattern();
$rfc4122Pattern = (new Rfc4122Validator())->getPattern();
```

The following *Uuid* methods are deprecated. If using these methods, you shouldn't have any problems on version 4, but you are encouraged to update your code, since they will go away in version 5.

- `getClockSeqHiAndReserved()`
- `getClockSeqLow()`
- `getClockSequence()`
- `getLeastSignificantBits()`
- `getMostSignificantBits()`
- `getNode()`
- `getTimeHiAndVersion()`
- `getTimeLow()`
- `getTimeMid()`
- `getTimestamp()`

Hint: There are no direct replacements for these methods. In ramsey/uuid version 3, they returned `int` or Moontoast\Math\BigNumber values, depending on the environment. To update your code, you should use the recommended alternates listed in *Deprecations: UuidInterface*, combined with the arbitrary-precision mathematics library of your choice (e.g., brick/math, gmp, bcmath, etc.).

Listing 38: Using brick/math to convert a node to a string integer

```
use Brick\Math\BigInteger;

$node = BigInteger::fromBase($uuid->getFields()->getNode()->toString(), 16);
```

Interface Changes

For those who customize ramsey/uuid by implementing the interfaces provided, there are a few breaking changes to note.

Hint: Most existing methods on interfaces have type hints added to them. If you implement any interfaces, please be aware of this and update your classes.

UuidInterface

Method	Description
__toString()	New method; returns `string`
`getDateTime()`	Deprecated; now returns DateTimeInterface
getFields()	Used to return `array`; now returns *Rfc4122\FieldsInterface*
getHex()	Used to return `string`; now returns *Type\Hexadecimal*
getInteger()	New method; returns *Type\Integer*

UuidFactoryInterface

Method	Description
uuid2()	New method; returns *Rfc4122\UuidV2*
uuid6()	New method; returns *Nonstandard\UuidV6*
fromDateTime()	New method; returns *UuidInterface*
fromInteger()	Changed to accept only strings
getValidator()	New method; returns *UuidInterface*

Builder\UuidBuilderInterface

Method	Description
`build()`	The second parameter used to accept `array $fields`; now accepts `string $bytes`

Converter\TimeConverterInterface

Method	Description
`calculateTime()`	Used to return `string[]`; now returns *`Type\Hexadecimal`*
`convertTime()`	New method; returns *`Type\Time`*

Provider\TimeProviderInterface

Method	Description
`currentTime()`	Method removed from interface; use `getTime()` instead
`getTime()`	New method; returns *`Type\Time`*

Provider\NodeProviderInterface

Method	Description
`getNode()`	Used to return `string|false|null`; now returns *`Type\Hexadecimal`*

Constructor Changes

There are a handful of constructor changes that might affect your use of ramsey/uuid, especially if you customize the library.

Uuid

The constructor for *`Ramsey\Uuid\Uuid`* is deprecated. However, there are a few changes to it that might affect your use of this class.

The first constructor parameter used to be `array $fields` and is now *`Rfc4122\FieldsInterface $fields`*.

`Converter\TimeConverterInterface $timeConverter` is required as a new fourth parameter.

Builder\DefaultUuidBuilder

While Builder\DefaultUuidBuilder is deprecated, it now inherits from Rfc4122\UuidBuilder, which requires `Converter\TimeConverterInterface $timeConverter` as its second constructor argument.

Provider\Node\FallbackNodeProvider

Provider\Node\FallbackNodeProvider now requires `iterable<Ramsey\Uuid\Provider\NodeProviderInterface>` as its constructor parameter.

```
use MyPackage\MyCustomNodeProvider;
use Ramsey\Uuid\Provider\Node\FallbackNodeProvider;
use Ramsey\Uuid\Provider\Node\RandomNodeProvider;
use Ramsey\Uuid\Provider\Node\SystemNodeProvider;

$nodeProviders = [];
$nodeProviders[] = new MyCustomNodeProvider();
$nodeProviders[] = new SystemNodeProvider();
$nodeProviders[] = new RandomNodeProvider();

$provider = new FallbackNodeProvider($nodeProviders);
```

Provider\Time\FixedTimeProvider

The constructor for Provider\Time\FixedTimeProvider no longer accepts an array. It accepts *`Type\Time`* instances.

1.8.2 Version 2 to 3

While we have made significant internal changes to the library, we have made every effort to ensure a seamless upgrade path from the 2.x series of this library to 3.x.

One major breaking change is the transition from the `Rhumsaa` root namespace to `Ramsey`. In most cases, all you will need is to change the namespace to `Ramsey` in your code, and everything will "just work."

Note: For more details on the namespace change, including reasons for the change, read the blog post "Introducing ramsey/uuid".

Here are full details on the breaking changes to the public API of this library:

1. All namespace references of `Rhumsaa` have changed to `Ramsey`. Simply change the namespace to `Ramsey` in your code and everything should work.
2. The console application has moved to ramsey/uuid-console. If using the console functionality, use Composer to require `ramsey/uuid-console`.
3. The Doctrine field type mapping has moved to ramsey/uuid-doctrine. If using the Doctrine functionality, use Composer to require `ramsey/uuid-doctrine`.

1.9 Frequently Asked Questions (FAQs)

- *How do I fix "rhumsaa/uuid is abandoned" messages?*
- *Why does ramsey/uuid use `final`?*

1.9.1 How do I fix "rhumsaa/uuid is abandoned" messages?

When installing your project's dependencies using Composer, you might see the following message:

```
Package rhumsaa/uuid is abandoned; you should avoid using it. Use
ramsey/uuid instead.
```

Don't panic. Simply execute the following commands with Composer:

```
composer remove rhumsaa/uuid
composer require ramsey/uuid=^2.9
```

After doing so, you will have the latest ramsey/uuid package in the 2.x series, and there will be no need to modify any code; the namespace in the 2.x series is still `Rhumsaa`.

1.9.2 Why does ramsey/uuid use `final`?

You might notice that many of the concrete classes returned in ramsey/uuid are marked as `final`. There are specific reasons for this choice, and I will offer a few solutions for those looking to extend or mock the classes for testing purposes.

But Why?

First, let's take a look at why ramsey/uuid uses `final`.

UUIDs are defined by a set of rules — published as RFC 4122 — and those rules shouldn't change. If they do, then it's no longer a UUID — at least not as defined by RFC 4122.

As an example, let's think about *`Rfc4122\UuidV1`*. If our application wants to do something special with this type, it might use the `instanceof` operator to check that a variable is a UuidV1, or it might use a type hint on a method argument. If a third-party library passes a UUID object to us that extends UuidV1 but overrides some very important internal logic, then we may no longer have a version 1 UUID. Perhaps we can all be adults and play nicely, but ramsey/uuid cannot make any guarantees for any subclasses of UuidV1.

However, ramsey/uuid *can* make guarantees about classes that implement *`UuidInterface`* or *`Rfc4122\UuidInterface`*.

So, if we're working with an instance of a class that is marked `final`, we can guarantee that the rules for the creation of that object will not change, even if a third-party library passes us an instance of the same class.

This is the reason why ramsey/uuid specifies certain *argument and return types* that are marked `final`. Since these are `final`, ramsey/uuid is able to guarantee the type of data these value objects contain. *`Type\Integer`* should never contain any characters other than numeral digits, and *`Type\Hexadecimal`* should never contain any characters other than hexadecimal digits. If other libraries could extend these and return them from UUID instances, then ramsey/uuid cannot guarantee their values.

This is very similar to using strict types with `int`, `float`, or `bool`. These types cannot change, so think of final classes in ramsey/uuid as types that cannot change.

Overriding Behavior

You may override the behavior of ramsey/uuid as much as you want. Despite the use of `final`, the library is very flexible. Take a look at the myriad opportunities to change how the library works:

- *Generating a Random Node*
- *Timestamp-first COMB Codec*
- *Replace the Default Factory*
- *And more...*

ramsey/uuid is able to provide this flexibility through the use of interfaces, factories, and dependency injection.

At the same time, ramsey/uuid is able to guarantee that neither a *`UuidV1`* nor a *`UuidV4`* nor an *`Integer`* nor a *`Time`*, etc. will ever change because of downstream code.

UUIDs have specific rules that make them practically unique. ramsey/uuid ensures that other code cannot change this expectation while allowing your code and third-party libraries to change how UUIDs are generated and to return different types of UUIDs not specified by RFC 4122.

Testing With UUIDs

Sometimes, the use of `final` can throw a wrench in our ability to write tests, but it doesn't have to be that way. To learn a few techniques for using ramsey/uuid instances in your tests, take a look at *Testing With UUIDs*.

1.10 Reference

1.10.1 Uuid

RamseyUuidUuid provides static methods for the most common functionality for generating and working with UUIDs. It also provides constants used throughout the ramsey/uuid library.

class Ramsey\Uuid**Uuid**

constant UUID_TYPE_TIME
Version 1: Gregorian Time UUID.

constant UUID_TYPE_DCE_SECURITY
Version 2: DCE Security UUID.

constant UUID_TYPE_HASH_MD5
Version 3: Name-based (MD5) UUID.

constant UUID_TYPE_RANDOM
Version 4: Random UUID.

constant UUID_TYPE_HASH_SHA1
Version 5: Name-based (SHA-1) UUID.

constant UUID_TYPE_REORDERED_TIME
Version 6: Reordered Time UUID.

constant UUID_TYPE_PEABODY
Deprecated. Use *Uuid::UUID_TYPE_REORDERED_TIME* instead.

constant NAMESPACE_DNS
The name string is a fully-qualified domain name.

constant NAMESPACE_URL
The name string is a URL.

constant NAMESPACE_OID
The name string is an ISO object identifier (OID).

constant NAMESPACE_X500
The name string is an X.500 DN in DER or a text output format.

constant NIL
The nil UUID is a special form of UUID that is specified to have all 128 bits set to zero.

constant DCE_DOMAIN_PERSON
DCE Security principal (person) domain.

constant DCE_DOMAIN_GROUP
DCE Security group domain.

constant DCE_DOMAIN_ORG
DCE Security organization domain.

constant RESERVED_NCS
Variant identifier: reserved, NCS backward compatibility.

constant RFC_4122
Variant identifier: the UUID layout specified in RFC 4122.

constant RESERVED_MICROSOFT
Variant identifier: reserved, Microsoft Corporation backward compatibility.

constant RESERVED_FUTURE
Variant identifier: reserved for future definition.

static uuid1 ([*$node* [, *$clockSeq*]])
Generates a version 1, Gregorian time UUID. See *Version 1: Gregorian Time*.

Parameters

- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A version 1 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV1*

static uuid2 (*$localDomain* [, *$localIdentifier* [, *$node* [, *$clockSeq*]]])
Generates a version 2, DCE Security UUID. See *Version 2: DCE Security*.

Parameters

- **$localDomain** (*int*) – The local domain to use (one of *Uuid::DCE_DOMAIN_PERSON*, *Uuid::DCE_DOMAIN_GROUP*, or *Uuid::DCE_DOMAIN_ORG*)
- **$localIdentifier** (*Ramsey\Uuid\Type\Integer|null*) – A local identifier for the domain (defaults to system UID or GID for *person* or *group*)

- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A version 2 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV2*

static uuid3 (*$ns*, *$name*)
Generates a version 3, name-based (MD5) UUID. See *Version 3: Name-based (MD5)*.

Parameters

- **$ns** (*Ramsey\Uuid\UuidInterface|string*) – The namespace for this identifier
- **$name** (*string*) – The name from which to generate an identifier

Returns A version 3 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV3*

static uuid4
Generates a version 4, random UUID. See *Version 4: Random*.

Returns A version 4 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV4*

static uuid5 (*$ns*, *$name*)
Generates a version 5, name-based (SHA-1) UUID. See *Version 5: Name-based (SHA-1)*.

Parameters

- **$ns** (*Ramsey\Uuid\UuidInterface|string*) – The namespace for this identifier
- **$name** (*string*) – The name from which to generate an identifier

Returns A version 5 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV5*

static uuid6 ([*$node*[, *$clockSeq*]])
Generates a version 6, reordered time UUID. See *Version 6: Reordered Time*.

Parameters

- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A version 6 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV6*

static fromString (*$uuid*)
Creates an instance of UuidInterface from the string standard representation.

Parameters

- **$uuid** (*string*) – The string standard representation of a UUID

Return type *Ramsey\Uuid\UuidInterface*

static fromBytes (*$bytes*)
Creates an instance of UuidInterface from a 16-byte string.

Parameters

- **$bytes** (*string*) – A 16-byte binary string representation of a UUID

Return type *Ramsey\Uuid\UuidInterface*

static fromInteger(*$integer*)
Creates an instance of UuidInterface from a 128-bit string integer.

Parameters

- **$integer** (*string*) – A 128-bit string integer representation of a UUID

Return type *Ramsey\Uuid\UuidInterface*

static fromDateTime(*$dateTime*[, *$node*[, *$clockSeq*]])
Creates a version 1 UUID instance from a DateTimeInterface instance.

Parameters

- **$dateTime** (*DateTimeInterface*) – The date from which to create the UUID instance
- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A version 1 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV1*

static isValid(*$uuid*)
Validates the string standard representation of a UUID.

Parameters

- **$uuid** (*string*) – The string standard representation of a UUID

Return type bool

static setFactory(*$factory*)
Sets the factory used to create UUIDs.

Parameters

- **$factory** (Ramsey\Uuid\UuidFactoryInterface) – A UUID factory to use for all UUID generation

Return type void

1.10.2 UuidInterface

interface Ramsey\Uuid**UuidInterface**
Represents a UUID.

compareTo(*$other*)

Parameters

- **$other** (Ramsey\Uuid\UuidInterface) – The UUID to compare

Returns Returns -1, 0, or 1 if the UUID is less than, equal to, or greater than the other UUID.

Return type int

equals(*$other*)

Parameters

- **$other** (*object|null*) – An object to test for equality with this UUID.

Returns Returns true if the UUID is equal to the provided object.

Return type bool

getBytes()

Returns A binary string representation of the UUID.

Return type string

getFields()

Returns The fields that comprise this UUID.

Return type *Ramsey\Uuid\Fields\FieldsInterface*

getHex()

Returns The hexadecimal representation of the UUID.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getInteger()

Returns The integer representation of the UUID.

Return type *Ramsey\Uuid\Type\Integer*

getUrn()

Returns The string standard representation of the UUID as a URN.

Return type string

toString()

Returns The string standard representation of the UUID.

Return type string

__toString()

Returns The string standard representation of the UUID.

Return type string

1.10.3 Fields\FieldsInterface

interface Ramsey\Uuid\Fields**FieldsInterface**

Represents the fields of a UUID.

getBytes()

Returns The bytes that comprise these fields.

Return type string

1.10.4 Rfc4122\UuidInterface

interface Ramsey\Uuid\Rfc4122**UuidInterface**
Implements *Ramsey\Uuid\UuidInterface*.

Rfc4122UuidInterface represents an RFC 4122 UUID. In addition to the methods defined on the interface, this interface additionally defines the following methods.

getFields()

Returns The fields that comprise this UUID.

Return type *Ramsey\Uuid\Rfc4122\FieldsInterface*

1.10.5 Rfc4122\FieldsInterface

interface Ramsey\Uuid\Rfc4122**FieldsInterface**
Implements *Ramsey\Uuid\Fields\FieldsInterface*.

Rfc4122FieldsInterface represents the fields of an RFC 4122 UUID. In addition to the methods defined on the interface, this class additionally defines the following methods.

getClockSeq()

Returns The full 16-bit clock sequence, with the variant bits (two most significant bits) masked out.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getClockSeqHiAndReserved()

Returns The high field of the clock sequence multiplexed with the variant.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getClockSeqLow()

Returns The low field of the clock sequence.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getNode()

Returns The node field.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getTimeHiAndVersion()

Returns The high field of the timestamp multiplexed with the version.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getTimeLow()

Returns The low field of the timestamp.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getTimeMid()

Returns The middle field of the timestamp.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getTimestamp()

Returns The full 60-bit timestamp, without the version.

Return type *Ramsey\Uuid\Type\Hexadecimal*

getVariant()
Returns the variant, which, for RFC 4122 variant UUIDs, should always be the value `2`.

Returns The UUID variant.

Return type `int`

getVersion()

Returns The UUID version.

Return type `int`

isNil()
A *nil* UUID is a special type of UUID with all 128 bits set to zero. Its string standard representation is always `00000000-0000-0000-0000-000000000000`.

Returns True if this UUID represents a nil UUID.

Return type `bool`

1.10.6 Rfc4122\UuidV1

class Ramsey\Uuid\Rfc4122**UuidV1**
Implements *Ramsey\Uuid\Rfc4122\UuidInterface*.

UuidV1 represents a *version 1, Gregorian time UUID*. In addition to providing the methods defined on the interface, this class additionally provides the following methods.

getDateTime()

Returns A date object representing the timestamp associated with the UUID.

Return type `\DateTimeInterface`

1.10.7 Rfc4122\UuidV2

class Ramsey\Uuid\Rfc4122**UuidV2**
Implements *Ramsey\Uuid\Rfc4122\UuidInterface*.

UuidV2 represents a *version 2, DCE Security UUID*. In addition to providing the methods defined on the interface, this class additionally provides the following methods.

getDateTime()
Returns a DateTimeInterface instance representing the timestamp associated with the UUID

> **Caution:** It is important to note that version 2 UUIDs suffer from some loss of timestamp precision. See *Lossy Timestamps* to learn more.

Returns A date object representing the timestamp associated with the UUID

Return type `\DateTimeInterface`

getLocalDomain()

Returns The local domain identifier for this UUID, which is one of *Ramsey\Uuid\Uuid::DCE_DOMAIN_PERSON*, *Ramsey\Uuid\Uuid::DCE_DOMAIN_GROUP*, or *Ramsey\Uuid\Uuid::DCE_DOMAIN_ORG*

Return type `int`

getLocalDomainName()

Returns A string name associated with the local domain identifier (one of "person," "group," or "org")

Return type `string`

getLocalIdentifier()

Returns The local identifier used when creating this UUID

Return type *Ramsey\Uuid\Type\Integer*

1.10.8 Rfc4122\UuidV3

class Ramsey\Uuid\Rfc4122**UuidV3**

Implements *Ramsey\Uuid\Rfc4122\UuidInterface*.

UuidV3 represents a *version 3, name-based (MD5) UUID*.

1.10.9 Rfc4122\UuidV4

class Ramsey\Uuid\Rfc4122**UuidV4**

Implements *Ramsey\Uuid\Rfc4122\UuidInterface*.

UuidV4 represents a *version 4, random UUID*.

1.10.10 Rfc4122\UuidV5

class Ramsey\Uuid\Rfc4122**UuidV5**

Implements *Ramsey\Uuid\Rfc4122\UuidInterface*.

UuidV5 represents a *version 5, name-based (SHA-1) UUID*.

1.10.11 Rfc4122\UuidV6

class Ramsey\Uuid\Rfc4122**UuidV6**

Implements *Ramsey\Uuid\Rfc4122\UuidInterface*.

UuidV6 represents a *version 6, reordered time UUID*. In addition to providing the methods defined on the interface, this class additionally provides the following methods.

getDateTime()

Returns A date object representing the timestamp associated with the UUID

Return type `\DateTimeInterface`

toUuidV1()

Returns A version 1 UUID, converted from this version 6 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV1*

static fromUuidV1

Parameters

- **$uuidV1** (Ramsey\Uuid\Rfc4122\UuidV1) – A version 1 UUID

Returns A version 6 UUID, converted from the given version 1 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV6*

1.10.12 Rfc4122\UuidV7

class Ramsey\Uuid\Rfc4122**UuidV7**

Implements *Ramsey\Uuid\Rfc4122\UuidInterface*.

UuidV7 represents a *version 7, Unix Epoch time UUID*. In addition to providing the methods defined on the interface, this class additionally provides the following methods.

getDateTime()

Returns A date object representing the timestamp associated with the UUID.

Return type \DateTimeInterface

1.10.13 Guid\Fields

class Ramsey\Uuid\Guid**Fields**

Implements *Ramsey\Uuid\Rfc4122\FieldsInterface*.

GuidFields represents the fields of a GUID.

1.10.14 Guid\Guid

class Ramsey\Uuid\Guid**Guid**

Implements *Ramsey\Uuid\UuidInterface*.

Guid represents a *Globally Unique Identifiers (GUIDs)*. In addition to providing the methods defined on the interface, this class additionally provides the following methods.

getFields()

Returns The fields that comprise this GUID.

Return type *Ramsey\Uuid\Guid\Fields*

1.10.15 Nonstandard\Fields

class Ramsey\Uuid\Nonstandard**Fields**

Implements *Ramsey\Uuid\Rfc4122\FieldsInterface*.

NonstandardFields represents the fields of a nonstandard UUID.

1.10.16 Nonstandard\Uuid

class Ramsey\Uuid\Nonstandard**Uuid**

Implements *Ramsey\Uuid\UuidInterface*.

NonstandardUuid represents *Other Nonstandard UUIDs*. In addition to providing the methods defined on the interface, this class additionally provides the following methods.

getFields()

Returns The fields that comprise this UUID

Return type *Ramsey\Uuid\Nonstandard\Fields*

1.10.17 Nonstandard\UuidV6

class Ramsey\Uuid\Nonstandard**UuidV6**

Attention: *Ramsey\Uuid\Nonstandard\UuidV6* is deprecated in favor of *Ramsey\Uuid\Rfc4122\UuidV6*. Please migrate any code using Nonstandard\UuidV6 to Rfc4122\UuidV6. The interface is otherwise identical.

Implements *Ramsey\Uuid\Rfc4122\UuidInterface*.

UuidV6 represents a *version 6, reordered time UUID*. In addition to providing the methods defined on the interface, this class additionally provides the following methods.

getDateTime()

Returns A date object representing the timestamp associated with the UUID

Return type \DateTimeInterface

toUuidV1()

Returns A version 1 UUID, converted from this version 6 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV1*

static fromUuidV1

Parameters

- **$uuidV1** (Ramsey\Uuid\Rfc4122\UuidV1) – A version 1 UUID

Returns A version 6 UUID, converted from the given version 1 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV6*

1.10.18 UuidFactoryInterface

interface Ramsey\Uuid**UuidFactoryInterface**

Represents a UUID factory.

getValidator()

Return type *Ramsey\Uuid\Validator\ValidatorInterface*

uuid1([*$node*[, *$clockSeq*]])

Generates a version 1, Gregorian time UUID. See *Version 1: Gregorian Time*.

Parameters

- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A version 1 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV1*

uuid2(*$localDomain*[, *$localIdentifier*[, *$node*[, *$clockSeq*]]])

Generates a version 2, DCE Security UUID. See *Version 2: DCE Security*.

Parameters

- **$localDomain** (*int*) – The local domain to use (one of *Uuid::DCE_DOMAIN_PERSON*, *Uuid::DCE_DOMAIN_GROUP*, or *Uuid::DCE_DOMAIN_ORG*)
- **$localIdentifier** (*Ramsey\Uuid\Type\Integer|null*) – A local identifier for the domain (defaults to system UID or GID for *person* or *group*)
- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A version 2 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV2*

uuid3(*$ns*, *$name*)

Generates a version 3, name-based (MD5) UUID. See *Version 3: Name-based (MD5)*.

Parameters

- **$ns** (*Ramsey\Uuid\UuidInterface|string*) – The namespace for this identifier
- **$name** (*string*) – The name from which to generate an identifier

Returns A version 3 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV3*

uuid4()

Generates a version 4, random UUID. See *Version 4: Random*.

Returns A version 4 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV4*

uuid5(*$ns*, *$name*)

Generates a version 5, name-based (SHA-1) UUID. See *Version 5: Name-based (SHA-1)*.

Parameters

- **$ns** (*Ramsey\Uuid\UuidInterface|string*) – The namespace for this identifier
- **$name** (*string*) – The name from which to generate an identifier

Returns A version 5 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV5*

uuid6 ([*$node* [, *$clockSeq*]])
Generates a version 6, reordered time UUID. See *Version 6: Reordered Time*.

Parameters

- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A version 6 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV6*

fromString (*$uuid*)
Creates an instance of UuidInterface from the string standard representation.

Parameters

- **$uuid** (*string*) – The string standard representation of a UUID

Return type *Ramsey\Uuid\UuidInterface*

fromBytes (*$bytes*)
Creates an instance of UuidInterface from a 16-byte string.

Parameters

- **$bytes** (*string*) – A 16-byte binary string representation of a UUID

Return type *Ramsey\Uuid\UuidInterface*

fromInteger (*$integer*)
Creates an instance of UuidInterface from a 128-bit string integer.

Parameters

- **$integer** (*string*) – A 128-bit string integer representation of a UUID

Return type *Ramsey\Uuid\UuidInterface*

fromDateTime (*$dateTime* [, *$node* [, *$clockSeq*]])
Creates a version 1 UUID instance from a DateTimeInterface instance.

Parameters

- **$dateTime** (*DateTimeInterface*) – The date from which to create the UUID instance
- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A version 1 UUID

Return type *Ramsey\Uuid\Rfc4122\UuidV1*

1.10.19 Types

class Ramsey\Uuid\Type**TypeInterface**
Implements JsonSerializable and Serializable.

TypeInterface ensures consistency in typed values returned by ramsey/uuid.

toString()

Return type string

__toString()

Return type string

class Ramsey\Uuid\Type**NumberInterface**
Implements *Ramsey\Uuid\Type\TypeInterface*.

NumberInterface ensures consistency in numeric values returned by ramsey/uuid.

isNegative()

Returns True if this number is less than zero, false otherwise.

Return type bool

class Ramsey\Uuid\Type**Decimal**
Implements *Ramsey\Uuid\Type\NumberInterface*.

A value object representing a decimal, for type-safety purposes, to ensure that decimals returned from ramsey/uuid methods as strings are truly decimals and not some other kind of string.

To support values as true decimals and not as floats or doubles, we store the decimals as strings.

class Ramsey\Uuid\Type**Hexadecimal**
Implements *Ramsey\Uuid\Type\TypeInterface*.

A value object representing a hexadecimal number, for type-safety purposes, to ensure that hexadecimal numbers returned from ramsey/uuid methods as strings are truly hexadecimal and not some other kind of string.

class Ramsey\Uuid\Type**Integer**
Implements *Ramsey\Uuid\Type\NumberInterface*.

A value object representing an integer, for type-safety purposes, to ensure that integers returned from ramsey/uuid methods as strings are truly integers and not some other kind of string.

To support large integers beyond PHP_INT_MAX and PHP_INT_MIN on both 64-bit and 32-bit systems, we store the integers as strings.

class Ramsey\Uuid\Type**Time**
Implements *Ramsey\Uuid\Type\TypeInterface*.

A value object representing a timestamp, for type-safety purposes, to ensure that timestamps used by ramsey/uuid are truly timestamp integers and not some other kind of string or integer.

getSeconds()

Return type *Ramsey\Uuid\Type\Integer*

getMicroseconds()

Return type *Ramsey\Uuid\Type\Integer*

1.10.20 Exceptions

All exceptions in the *Ramsey\Uuid* namespace implement *Ramsey\Uuid\Exception\UuidExceptionInterface*. This provides a base type you may use to catch any and all exceptions that originate from this library.

interface Ramsey\Uuid\Exception**UuidExceptionInterface**
This is the interface all exceptions in ramsey/uuid must implement.

exception Ramsey\Uuid\Exception**BuilderNotFoundException**
Extends RuntimeException.

Thrown to indicate that no suitable UUID builder could be found.

exception Ramsey\Uuid\Exception**DateTimeException**
Extends RuntimeException.

Thrown to indicate that the PHP DateTime extension encountered an exception or error.

exception Ramsey\Uuid\Exception**DceSecurityException**
Extends RuntimeException.

Thrown to indicate an exception occurred while dealing with DCE Security (version 2) UUIDs

exception Ramsey\Uuid\Exception**InvalidArgumentException**
Extends InvalidArgumentException.

Thrown to indicate that the argument received is not valid.

exception Ramsey\Uuid\Exception**InvalidBytesException**
Extends RuntimeException.

Thrown to indicate that the bytes being operated on are invalid in some way.

exception Ramsey\Uuid\Exception**InvalidUuidStringException**
Extends *Ramsey\Uuid\Exception\InvalidArgumentException*.

Thrown to indicate that the string received is not a valid UUID.

exception Ramsey\Uuid\Exception**NameException**
Extends RuntimeException.

Thrown to indicate that an error occurred while attempting to hash a namespace and name

exception Ramsey\Uuid\Exception**NodeException**
Extends RuntimeException.

Thrown to indicate that attempting to fetch or create a node ID encountered an error.

exception Ramsey\Uuid\Exception**RandomSourceException**
Extends RuntimeException.

Thrown to indicate that the source of random data encountered an error.

exception Ramsey\Uuid\Exception**TimeSourceException**
Extends RuntimeException.

Thrown to indicate that the source of time encountered an error.

exception Ramsey\Uuid\Exception**UnableToBuildUuidException**
Extends RuntimeException.

Thrown to indicate a builder is unable to build a UUID.

exception Ramsey\Uuid\Exception**UnsupportedOperationException**
Extends LogicException.

Thrown to indicate that the requested operation is not supported.

1.10.21 Helper Functions

ramsey/uuid additionally provides the following helper functions, which return only the string standard representation of a UUID.

Ramsey\Uuid\v1([$node[, $clockSeq]])
Generates a string standard representation of a version 1, Gregorian time UUID.

Parameters

- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A string standard representation of a version 1 UUID

Return type string

Ramsey\Uuid\v2($localDomain[, $localIdentifier[, $node[, $clockSeq]]])
Generates a string standard representation of a version 2, DCE Security UUID.

Parameters

- **$localDomain** (*int*) – The local domain to use (one of Uuid::DCE_DOMAIN_PERSON, Uuid::DCE_DOMAIN_GROUP, or Uuid::DCE_DOMAIN_ORG)
- **$localIdentifier** (*Ramsey\Uuid\Type\Integer|null*) – A local identifier for the domain (defaults to system UID or GID for *person* or *group*)
- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A string standard representation of a version 2 UUID

Return type string

Ramsey\Uuid\v3($ns, $name)
Generates a string standard representation of a version 3, name-based (MD5) UUID.

Parameters

- **$ns** (*Ramsey\Uuid\UuidInterface|string*) – The namespace for this identifier
- **$name** (*string*) – The name from which to generate an identifier

Returns A string standard representation of a version 3 UUID

Return type string

Ramsey\Uuid\v4()
Generates a string standard representation of a version 4, random UUID.

Returns A string standard representation of a version 4 UUID

Return type string

Ramsey\Uuid\v5($ns, $name)
Generates a string standard representation of a version 5, name-based (SHA-1) UUID.

Parameters

- **$ns** (*Ramsey\Uuid\UuidInterface|string*) – The namespace for this identifier
- **$name** (*string*) – The name from which to generate an identifier

Returns A string standard representation of a version 5 UUID

Return type string

Ramsey\Uuid\v6([$node[, $clockSeq]])
Generates a string standard representation of a version 6, reordered time UUID.

Parameters

- **$node** (*Ramsey\Uuid\Type\Hexadecimal|null*) – An optional hexadecimal node to use
- **$clockSeq** (*int|null*) – An optional clock sequence to use

Returns A string standard representation of a version 6 UUID

Return type string

1.10.22 Predefined Namespaces

RFC 4122 defines a handful of UUIDs to use with "for some potentially interesting name spaces."

Constant	Description
Uuid::NAMESPACE_DNS	The name string is a fully-qualified domain name.
Uuid::NAMESPACE_URL	The name string is a URL.
Uuid::NAMESPACE_OID	The name string is an ISO object identifier (OID).
Uuid::NAMESPACE_X500	The name string is an X.500 DN in DER or a text output format.

1.10.23 Calculators

interface Ramsey\Uuid\Math**CalculatorInterface**
Provides functionality for performing mathematical calculations.

add(*$augend*, *...$addends*)

Parameters

- **$augend** (Ramsey\Uuid\Type\NumberInterface) – The first addend (the integer being added to)
- **...$addends** (Ramsey\Uuid\Type\NumberInterface) – The additional integers to a add to the augend

Returns The sum of all the parameters

Return type *Ramsey\Uuid\Type\NumberInterface*

subtract(*$minuend*, *...$subtrahends*)

Parameters

- **$minuend** (Ramsey\Uuid\Type\NumberInterface) – The integer being subtracted from

- **...$subtrahends** (Ramsey\Uuid\Type\NumberInterface) – The integers to subtract from the minuend

Returns The difference after subtracting all parameters

Return type *Ramsey\Uuid\Type\NumberInterface*

multiply (*$multiplicand*, *...$multipliers*)

Parameters

- **$multiplicand** (Ramsey\Uuid\Type\NumberInterface) – The integer to be multiplied
- **...$multipliers** (Ramsey\Uuid\Type\NumberInterface) – The factors by which to multiply the multiplicand

Returns The product of multiplying all the provided parameters

Return type *Ramsey\Uuid\Type\NumberInterface*

divide (*$roundingMode*, *$scale*, *$dividend*, *...$divisors*)

Parameters

- **$roundingMode** (*int*) – The strategy for rounding the quotient; one of the *Ramsey\Uuid\Math\RoundingMode* constants
- **$scale** (*int*) – The scale to use for the operation
- **$dividend** (Ramsey\Uuid\Type\NumberInterface) – The integer to be divided
- **...$divisors** (Ramsey\Uuid\Type\NumberInterface) – The integers to divide `$dividend` by, in the order in which the division operations should take place (left-to-right)

Returns The quotient of dividing the provided parameters left-to-right

Return type *Ramsey\Uuid\Type\NumberInterface*

fromBase (*$value*, *$base*)

Converts a value from an arbitrary base to a base-10 integer value.

Parameters

- **$value** (*string*) – The value to convert
- **$base** (*int*) – The base to convert from (i.e., 2, 16, 32, etc.)

Returns The base-10 integer value of the converted value

Return type *Ramsey\Uuid\Type\Integer*

toBase (*$value*, *$base*)

Converts a base-10 integer value to an arbitrary base.

Parameters

- **$value** (Ramsey\Uuid\Type\Integer) – The integer value to convert
- **$base** (*int*) – The base to convert to (i.e., 2, 16, 32, etc.)

Returns The value represented in the specified base

Return type `string`

toHexadecimal(*$value*)
Converts an Integer instance to a Hexadecimal instance.

Parameters

- **$value** (Ramsey\Uuid\Type\Integer) – The Integer to convert to Hexadecimal

Return type *Ramsey\Uuid\Type\Hexadecimal*

toInteger(*$value*)
Converts a Hexadecimal instance to an Integer instance.

Parameters

- **$value** (Ramsey\Uuid\Type\Hexadecimal) – The Hexadecimal to convert to Integer

Return type *Ramsey\Uuid\Type\Integer*

class Ramsey\Uuid\Math**RoundingMode**

constant UNNECESSARY
Asserts that the requested operation has an exact result, hence no rounding is necessary.

constant UP
Rounds away from zero.

Always increments the digit prior to a nonzero discarded fraction. Note that this rounding mode never decreases the magnitude of the calculated value.

constant DOWN
Rounds towards zero.

Never increments the digit prior to a discarded fraction (i.e., truncates). Note that this rounding mode never increases the magnitude of the calculated value.

constant CEILING
Rounds towards positive infinity.

If the result is positive, behaves as for *UP*; if negative, behaves as for *DOWN*. Note that this rounding mode never decreases the calculated value.

constant FLOOR
Rounds towards negative infinity.

If the result is positive, behave as for *DOWN*; if negative, behave as for *UP*. Note that this rounding mode never increases the calculated value.

constant HALF_UP
Rounds towards "nearest neighbor" unless both neighbors are equidistant, in which case round up.

Behaves as for *UP* if the discarded fraction is >= 0.5; otherwise, behaves as for *DOWN*. Note that this is the rounding mode commonly taught at school.

constant HALF_DOWN
Rounds towards "nearest neighbor" unless both neighbors are equidistant, in which case round down.

Behaves as for *UP* if the discarded fraction is > 0.5; otherwise, behaves as for *DOWN*.

constant HALF_CEILING
Rounds towards "nearest neighbor" unless both neighbors are equidistant, in which case round towards positive infinity.

If the result is positive, behaves as for *HALF_UP*; if negative, behaves as for *HALF_DOWN*.

constant HALF_FLOOR
Rounds towards "nearest neighbor" unless both neighbors are equidistant, in which case round towards negative infinity.

If the result is positive, behaves as for *HALF_DOWN*; if negative, behaves as for *HALF_UP*.

constant HALF_EVEN
Rounds towards the "nearest neighbor" unless both neighbors are equidistant, in which case rounds towards the even neighbor.

Behaves as for *HALF_UP* if the digit to the left of the discarded fraction is odd; behaves as for *HALF_DOWN* if it's even.

Note that this is the rounding mode that statistically minimizes cumulative error when applied repeatedly over a sequence of calculations. It is sometimes known as "Banker's rounding", and is chiefly used in the USA.

1.10.24 Validators

interface Ramsey\Uuid\Validator**ValidatorInterface**

getPattern()

Returns The regular expression pattern used by this validator

Return type string

validate(*$uuid*)

Parameters

- **$uuid** (*string*) – The string to validate as a UUID

Returns True if the provided string represents a UUID, false otherwise

Return type bool

class Ramsey\Uuid\Validator**GenericValidator**
Implements *Ramsey\Uuid\Validator\ValidatorInterface*.

GenericValidator validates strings as UUIDs of any variant.

class Ramsey\Uuid\Rfc4122**Validator**
Implements *Ramsey\Uuid\Validator\ValidatorInterface*.

Rfc4122Validator validates strings as UUIDs of the RFC 4122 variant.

1.11 Copyright

1.12 ramsey/uuid for Enterprise

Available as part of the Tidelift Subscription

Tidelift is working with the maintainers of ramsey/uuid and thousands of other open source projects to deliver commercial support and maintenance for the open source dependencies you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact dependencies you use.

Learn More Request a Demo

Enterprise-ready open source software — managed for you

The Tidelift Subscription is a managed open source subscription for application dependencies covering millions of open source projects across JavaScript, Python, Java, PHP, Ruby, .NET, and more.

Your subscription includes:

Security updates Tidelift's security response team coordinates patches for new breaking security vulnerabilities and alerts immediately through a private channel, so your software supply chain is always secure.

Licensing verification and indemnification Tidelift verifies license information to enable easy policy enforcement and adds intellectual property indemnification to cover creators and users in case something goes wrong. You always have a 100% up-to-date bill of materials for your dependencies to share with your legal team, customers, or partners.

Maintenance and code improvement Tidelift ensures the software you rely on keeps working as long as you need it to work. Your managed dependencies are actively maintained and we recruit additional maintainers where required.

Package selection and version guidance We help you choose the best open source packages from the start—and then guide you through updates to stay on the best releases as new issues arise.

Roadmap input Take a seat at the table with the creators behind the software you use. Tidelift's participating maintainers earn more income as their software is used by more subscribers, so they're interested in knowing what you need.

Tooling and cloud integration Tidelift works with GitHub, GitLab, BitBucket, and more. We support every cloud platform (and other deployment targets, too).

The end result? All of the capabilities you expect from commercial-grade software, for the full breadth of open source you use. That means less time grappling with esoteric open source trivia, and more time building your own applications—and your business.

Learn More Request a Demo

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

PHP NAMESPACE INDEX

r

Ramsey\Uuid, 47
Ramsey\Uuid\Exception, 57
Ramsey\Uuid\Fields, 48
Ramsey\Uuid\Guid, 52
Ramsey\Uuid\Math, 59
Ramsey\Uuid\Nonstandard, 53
Ramsey\Uuid\Rfc4122, 62
Ramsey\Uuid\Type, 56
Ramsey\Uuid\Validator, 62

INDEX

Symbols

__toString() (*Ramsey\Uuid\Type\TypeInterface method*), 56
__toString() (*Ramsey\Uuid\UuidInterface method*), 48

A

add() (*Ramsey\Uuid\Math\CalculatorInterface method*), 59

B

BuilderNotFoundException, 57

C

CalculatorInterface (*interface in Ramsey\Uuid\Math*), 59
compareTo() (*Ramsey\Uuid\UuidInterface method*), 47

D

DateTimeException, 57
DceSecurityException, 57
Decimal (*class in Ramsey\Uuid\Type*), 56
divide() (*Ramsey\Uuid\Math\CalculatorInterface method*), 60

E

equals() (*Ramsey\Uuid\UuidInterface method*), 47

F

Fields (*class in Ramsey\Uuid\Guid*), 52
Fields (*class in Ramsey\Uuid\Nonstandard*), 52
FieldsInterface (*interface in Ramsey\Uuid\Fields*), 48
FieldsInterface (*interface in Ramsey\Uuid\Rfc4122*), 49
fromBase() (*Ramsey\Uuid\Math\CalculatorInterface method*), 60
fromBytes() (*Ramsey\Uuid\Uuid method*), 46
fromBytes() (*Ramsey\Uuid\UuidFactoryInterface method*), 55
fromDateTime() (*Ramsey\Uuid\Uuid method*), 47
fromDateTime() (*Ramsey\Uuid\UuidFactoryInterface method*), 55
fromInteger() (*Ramsey\Uuid\Uuid method*), 47
fromInteger() (*Ramsey\Uuid\UuidFactoryInterface method*), 55
fromString() (*Ramsey\Uuid\Uuid method*), 46
fromString() (*Ramsey\Uuid\UuidFactoryInterface method*), 55
fromUuidV1() (*Ramsey\Uuid\Nonstandard\UuidV6 method*), 53
fromUuidV1() (*Ramsey\Uuid\Rfc4122\UuidV6 method*), 51

G

GenericValidator (*class in Ramsey\Uuid\Validator*), 62
getBytes() (*Ramsey\Uuid\Fields\FieldsInterface method*), 48
getBytes() (*Ramsey\Uuid\UuidInterface method*), 48
getClockSeq() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 49
getClockSeqHiAndReserved() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 49
getClockSeqLow() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 49
getDateTime() (*Ramsey\Uuid\Nonstandard\UuidV6 method*), 53
getDateTime() (*Ramsey\Uuid\Rfc4122\UuidV1 method*), 50
getDateTime() (*Ramsey\Uuid\Rfc4122\UuidV2 method*), 50
getDateTime() (*Ramsey\Uuid\Rfc4122\UuidV6 method*), 51
getDateTime() (*Ramsey\Uuid\Rfc4122\UuidV7 method*), 52
getFields() (*Ramsey\Uuid\Guid\Guid method*), 52
getFields() (*Ramsey\Uuid\Nonstandard\Uuid*

method), 53
getFields() (*Ramsey\Uuid\Rfc4122\UuidInterface method*), 49
getFields() (*Ramsey\Uuid\UuidInterface method*), 48
getHex() (*Ramsey\Uuid\UuidInterface method*), 48
getInteger() (*Ramsey\Uuid\UuidInterface method*), 48
getLocalDomain() (*Ramsey\Uuid\Rfc4122\UuidV2 method*), 50
getLocalDomainName() (*Ramsey\Uuid\Rfc4122\UuidV2 method*), 51
getLocalIdentifier() (*Ramsey\Uuid\Rfc4122\UuidV2 method*), 51
getMicroseconds() (*Ramsey\Uuid\Type\Time method*), 56
getNode() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 49
getPattern() (*Ramsey\Uuid\Validator\ValidatorInterface method*), 62
getSeconds() (*Ramsey\Uuid\Type\Time method*), 56
getTimeHiAndVersion() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 49
getTimeLow() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 49
getTimeMid() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 49
getTimestamp() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 49
getUrn() (*Ramsey\Uuid\UuidInterface method*), 48
getValidator() (*Ramsey\Uuid\UuidFactoryInterface method*), 54
getVariant() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 50
getVersion() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 50
Guid (*class in Ramsey\Uuid\Guid*), 52

H

Hexadecimal (*class in Ramsey\Uuid\Type*), 56

I

Integer (*class in Ramsey\Uuid\Type*), 56
InvalidArgumentException, 57
InvalidBytesException, 57
InvalidUuidStringException, 57
isNegative() (*Ramsey\Uuid\Type\NumberInterface method*), 56
isNil() (*Ramsey\Uuid\Rfc4122\FieldsInterface method*), 50
isValid() (*Ramsey\Uuid\Uuid method*), 47

M

multiply() (*Ramsey\Uuid\Math\CalculatorInterface method*), 60

N

NameException, 57
NodeException, 57
NumberInterface (*class in Ramsey\Uuid\Type*), 56

R

Ramsey\Uuid (*namespace*), 44, 47, 54
Ramsey\Uuid\Exception (*namespace*), 57
Ramsey\Uuid\Fields (*namespace*), 48
Ramsey\Uuid\Guid (*namespace*), 52
Ramsey\Uuid\Math (*namespace*), 59
Ramsey\Uuid\Nonstandard (*namespace*), 52, 53
Ramsey\Uuid\Rfc4122 (*namespace*), 49–52, 62
Ramsey\Uuid\Type (*namespace*), 56
Ramsey\Uuid\Validator (*namespace*), 62
RandomSourceException, 57
RoundingMode (*class in Ramsey\Uuid\Math*), 61
RoundingMode::CEILING (*class constant*), 61
RoundingMode::DOWN (*class constant*), 61
RoundingMode::FLOOR (*class constant*), 61
RoundingMode::HALF_CEILING (*class constant*), 61
RoundingMode::HALF_DOWN (*class constant*), 61
RoundingMode::HALF_EVEN (*class constant*), 62
RoundingMode::HALF_FLOOR (*class constant*), 61
RoundingMode::HALF_UP (*class constant*), 61
RoundingMode::UNNECESSARY (*class constant*), 61
RoundingMode::UP (*class constant*), 61

S

setFactory() (*Ramsey\Uuid\Uuid method*), 47
subtract() (*Ramsey\Uuid\Math\CalculatorInterface method*), 59

T

Time (*class in Ramsey\Uuid\Type*), 56
TimeSourceException, 57
toBase() (*Ramsey\Uuid\Math\CalculatorInterface method*), 60
toHexadecimal() (*Ramsey\Uuid\Math\CalculatorInterface method*), 60

toInteger() (*Ramsey\Uuid\Math\CalculatorInterface method*), 61
toString() (*Ramsey\Uuid\Type\TypeInterface method*), 56
toString() (*Ramsey\Uuid\UuidInterface method*), 48
toUuidV1() (*Ramsey\Uuid\Nonstandard\UuidV6 method*), 53
toUuidV1() (*Ramsey\Uuid\Rfc4122\UuidV6 method*), 51
TypeInterface (*class in Ramsey\Uuid\Type*), 56

U

UnableToBuildUuidException, 57
UnsupportedOperationException, 57
Uuid (*class in Ramsey\Uuid*), 44
Uuid (*class in Ramsey\Uuid\Nonstandard*), 53
uuid1() (*Ramsey\Uuid\Uuid method*), 45
uuid1() (*Ramsey\Uuid\UuidFactoryInterface method*), 54
uuid2() (*Ramsey\Uuid\Uuid method*), 45
uuid2() (*Ramsey\Uuid\UuidFactoryInterface method*), 54
uuid3() (*Ramsey\Uuid\Uuid method*), 46
uuid3() (*Ramsey\Uuid\UuidFactoryInterface method*), 54
uuid4() (*Ramsey\Uuid\Uuid method*), 46
uuid4() (*Ramsey\Uuid\UuidFactoryInterface method*), 54
uuid5() (*Ramsey\Uuid\Uuid method*), 46
uuid5() (*Ramsey\Uuid\UuidFactoryInterface method*), 54
uuid6() (*Ramsey\Uuid\Uuid method*), 46
uuid6() (*Ramsey\Uuid\UuidFactoryInterface method*), 55
Uuid::DCE_DOMAIN_GROUP (*class constant*), 45
Uuid::DCE_DOMAIN_ORG (*class constant*), 45
Uuid::DCE_DOMAIN_PERSON (*class constant*), 45
Uuid::NAMESPACE_DNS (*class constant*), 45
Uuid::NAMESPACE_OID (*class constant*), 45
Uuid::NAMESPACE_URL (*class constant*), 45
Uuid::NAMESPACE_X500 (*class constant*), 45
Uuid::NIL (*class constant*), 45
Uuid::RESERVED_FUTURE (*class constant*), 45
Uuid::RESERVED_MICROSOFT (*class constant*), 45
Uuid::RESERVED_NCS (*class constant*), 45
Uuid::RFC_4122 (*class constant*), 45
Uuid::UUID_TYPE_DCE_SECURITY (*class constant*), 44
Uuid::UUID_TYPE_HASH_MD5 (*class constant*), 44
Uuid::UUID_TYPE_HASH_SHA1 (*class constant*), 44
Uuid::UUID_TYPE_PEABODY (*class constant*), 44
Uuid::UUID_TYPE_RANDOM (*class constant*), 44
Uuid::UUID_TYPE_REORDERED_TIME (*class constant*), 44
Uuid::UUID_TYPE_TIME (*class constant*), 44
UuidExceptionInterface (*interface in Ramsey\Uuid\Exception*), 57
UuidFactoryInterface (*interface in Ramsey\Uuid*), 54
UuidInterface (*interface in Ramsey\Uuid*), 47
UuidInterface (*interface in Ramsey\Uuid\Rfc4122*), 49
UuidV1 (*class in Ramsey\Uuid\Rfc4122*), 50
UuidV2 (*class in Ramsey\Uuid\Rfc4122*), 50
UuidV3 (*class in Ramsey\Uuid\Rfc4122*), 51
UuidV4 (*class in Ramsey\Uuid\Rfc4122*), 51
UuidV5 (*class in Ramsey\Uuid\Rfc4122*), 51
UuidV6 (*class in Ramsey\Uuid\Nonstandard*), 53
UuidV6 (*class in Ramsey\Uuid\Rfc4122*), 51
UuidV7 (*class in Ramsey\Uuid\Rfc4122*), 52

V

validate() (*Ramsey\Uuid\Validator\ValidatorInterface method*), 62
Validator (*class in Ramsey\Uuid\Rfc4122*), 62
ValidatorInterface (*interface in Ramsey\Uuid\Validator*), 62