encoding/json/v2 Migration Guide

- Why migrate?
- API vs behavior changes
- Options
- Migration
  - All-at-once {#all-at-once}
  - Option-by-option {#option-by-option}
  - jsonsplit {#jsonsplit}

Introduced in Go 1.27, the encoding/json/v2 package is a major revision of the original encoding/json package. This guide describes why you may want to migrate from the v1 to v2 package, and the mechanics of doing so safely.

Why migrate?

First things first: you don't have to! The `encoding/json` package will never go away. It is covered by the Go 1 compatibility promise, so packages using `encoding/json` will continue working indefinitely.

In addition, `encoding/json` and `encoding/json/v2` are compatible with each other. For example, if a type implements custom marshaling behavior with `encoding/json/v2.MarshalerTo`, the callers marshaling this type through `encoding/json.Marshal` will still go through the custom marshaler. Similarly, new `json` struct tags introduced in Go 1.27 alongside `encoding/json/v2` are also supported by `encoding/json`.

Though you are not required to migrate, there are several good reasons to do so:

First, the new API is easier to use. For example, easily marshal to an `io.Writer` with `encoding/json/v2.MarshalWrite`, rather than needing an `encoding/json.Encoder`. `encoding/json/v2.Marshalers` allows overriding the marshal behavior of specific types, even if you

do not control those types. `encoding/json/v2.MatchCaseInsenstiveNames` allows control over the case sensitivity of matching JSON object member names to Go struct fields.

While these improvements are nice, the best reason to migrate is that the v2 package chooses stricter, more interoperable defaults than v1. The `encoding/json` documentation contains the complete set of differences, but some highlights include:

- In v1, bytes of invalid UTF-8 within a string are silently replaced with the Unicode replacement character. In contrast, in v2 the presence of invalid UTF-8 results in an error.
- In v1, a JSON object with duplicate names is permitted. In contrast, in v2 a JSON object with duplicate names results in an error.
- In v1, a nil Go slice or Go map is marshaled as a JSON null. In contrast, v2 marshals a nil Go slice or Go map as an empty JSON array or JSON object, respectively.
- In v1, errors are never reported at runtime for Go struct types that have some form of structural error (e.g., a malformed field tag). In contrast, v2 reports a runtime error for Go types that are invalid as they relate to JSON serialization.

These changes are designed to make `encoding/json/v2` more interoperable with the wider JSON ecosystem, less surprising, and less error-prone. But they are not backwards compatible; some applications may depend on the v1 behavior. Therefore, migration to v2 must involve careful testing to ensure compatibility.

API vs behavior changes

For simple marshaling and unmarshaling, the Go APIs are largely compatible at the language level, and thus the API migration is trivial. For example, `b, err := json.Marshal(v)` will continue to compile simply by switching `import "encoding/json"` to `import "encoding/json/v2"`.

The difficult part of migrating from v1 to v2 is not the API differences, but behavior changes in marshaling and unmarshaling. Consider this program:

```
package main

import (
	"encoding/json"
	"fmt"
)

type Pet struct {
	Name      string
	Nicknames []string
```

```
}

func main() {
	pets := []Pet{
		{Name: "Oliver", Nicknames: []string{"Ollie", "Olliepop"}},
		{Name: "Remi"},
	}
	b, err := json.Marshal(pets)
	if err != nil {
		panic(err)
	}
	fmt.Println(string(b))
}
```

When run, this outputs:

```
[{"Name":"Oliver","Nicknames":["Ollie","Olliepop"]},{"Name":"Remi","Nicknames":null}]
```

If we trivially migrate this program to `encoding/json/v2` by changing the import, the program still compiles. When run, it outputs:

```
[{"Name":"Oliver","Nicknames":["Ollie","Olliepop"]},{"Name":"Remi","Nicknames":[]}]
```

Notice that the "Nicknames" field for Remi has changed from `null` to `[]`. If this were a new program, using an empty array is likely a nice improvement, but in an existing application downstream consumers of this output may be depending on the presence of `null`, so this change may break them.

Options

All of the places where v2 behavior diverges from v1 are covered by `Options` which allow specifying the v1 behavior using the v2 API. To use the v2 API, but specify all v1 behaviors, use `DefaultOptionsV1`:

```
package main

import (
	jsonv1 "encoding/json"
	"encoding/json/v2"
```

```
	"fmt"
)

type Pet struct {
	Name      string
	Nicknames []string
}

func main() {
	pets := []Pet{
		{Name: "Oliver", Nicknames: []string{"Ollie", "Olliepop"}},
		{Name: "Remi"},
	}
	b, err := json.Marshal(pets, jsonv1.DefaultOptionsV1())
	if err != nil {
		panic(err)
	}
	fmt.Println(string(b))
}
```

When run, this once again outputs `null`:

```
[{"Name":"Oliver","Nicknames":["Ollie","Olliepop"]},{"Name":"Remi","Nicknames":null}]
```

The DefaultOptionsV1 documentation lists the full set of options used for v1 compatibility. Later options override earlier options, so you can use this list to enable v2 behaviors one at a time.

Migration

Depending on the type of application and its risk tolerance, there are several different ways to approach a v2 migration:

- High risk tolerance: all-at-once
- Low risk tolerance or troubleshooting: option-by-option
- Production server applications: jsonsplit

All-at-once {#all-at-once}

If the application is simple or has high risk tolerance (there is little consequence to the migration causing problems), then there may be no need for an elaborate migration process. Simply update callsites to use `encoding/json/v2`, make sure the tests pass, and check it in.

If you do run into compatibility issues, the source of the difference may be clear from the change or error message, in which case you can set the appropriate compatibility option. If the source of the problem is not clear, you may want to use one of the approaches below to help troubleshoot.

This can also be a quick way to find and fix obvious issues (such as those identified by unit tests) before moving on to a more nuanced approach to track down the remainder.

Option-by-option {#option-by-option}

If the application is complex or has a low risk tolerance, then you may need to take a slower, more careful approach.

As mentioned above, calling `Marshal` or `Unmarshal` with DefaultOptionsV1 makes the call behave identically to `encoding/json`. As a first step, migrate all calls to `encoding/json/v2` with `DefaultOptionsV1`. This is a trivial and safe change; in fact, this is exactly how `encoding/json` implements `Marshal` and `Unmarshal`!

Additional passed options override earlier options, so you can disable individual v1 compatibility options. For example, `json.Marshal(v, jsonv1.DefaultOptionsV1(), json.FormatNilSliceAsNull(false))` will behave like v1 except that `nil` slices format as empty arrays.

This provides a path to perform a slower migration rather than changing all behavior at once. You could enable one option at a time to be very sure what is changing, or group similar options. Alternatively, when troubleshooting these options provide a way to bisect down to the exact breaking behavior change, similar to the option detection we'll see in `jsonsplit` below.

jsonsplit {#jsonsplit}

`github.com/go-json-experiment/jsonsplit` is a JSON wrapper package that aids migration by reporting differences between v1 and v2 in a production setting.

`jsonsplit` provides `Marshal` and `Unmarshal` wrapper functions which behave the same as `encoding/json` by default, but can be configured at runtime to use v1, v2, or both.

When configured to use both (`CallBothButReturnV1`), `jsonsplit.Marshal` will marshal the input twice: once with v1 and v2. It will report any differences, but still return the v1 value to the caller. This allows a production service to report differences without changing its behavior. With the optional `AutoDetectOptions`, `jsonsplit` will even automatically determine which specific options cause the difference.

Note that this functionality comes at a cost. Marshaling with both v1 and v2 to detect differences will approximately double the cost of marshaling, and `AutoDetectOptions` performs even more marshals to narrow down the relevant options. To mitigate these costs, `jsonsplit` allows checking only a random subset of calls via `SetMarshalCallRatio`.

Here, we have applied `jsonsplit` to the original example:

```
package main

import (
	"fmt"

	"github.com/go-json-experiment/jsonsplit"
)

func init() {
	// Call both v1 and v2 so we can detect differences, but continue using
	// v1 output.
	jsonsplit.GlobalCodec.SetMarshalCallMode(jsonsplit.CallBothButReturnV1)

	// Specify that when a difference is detected, to auto-detect which
	// options are causing the difference.
	jsonsplit.GlobalCodec.AutoDetectOptions = true

	// Log every time we detect a difference between v1 and v2.
	jsonsplit.GlobalCodec.ReportDifference = func(d jsonsplit.Difference) {
		fmt.Printf("detected jsonv1-to-jsonv2 difference: %v\n", d)
	}
}
```

```
type Pet struct {
	Name      string
	Nicknames []string
}

func main() {
	pets := []Pet{
		{Name: "Oliver", Nicknames: []string{"Ollie", "Olliepop"}},
		{Name: "Remi"},
	}
	b, err := jsonsplit.Marshal(pets)
	if err != nil {
		panic(err)
	}
	fmt.Println(string(b))
}
```

When run, this reports the difference and even that `FormatNilSliceAsNull` is responsible for the difference:

```
detected jsonv1-to-jsonv2 difference:
{"Caller":"main.main+5","Func":"Marshal","GoType":"[]main.Pet","JSONValueV1":[{"Name":"Oliver"
,"Nicknames":["Ollie","Olliepop"]},{"Name":"Remi","Nicknames":null}],"JSONValueV2":[{"Name":"O
liver","Nicknames":["Ollie","Olliepop"]},{"Name":"Remi","Nicknames":[]}],"Options":["jsonv2.Fo
rmatNilSliceAsNull"]}
[{"Name":"Oliver","Nicknames":["Ollie","Olliepop"]},{"Name":"Remi","Nicknames":null}]
```

We can migrate our production service smoothly using a procedure like the following:

1. Switch callsites to `jsonsplit`, set `CallBothButReturnV1`, `AutoDetectOptions` (optional), and wire up your preferred monitoring approach to `ReportDifference` (such as logging or published metrics).
2. Monitor your production environment for reported differences.
3. Encode differences.

Where `jsonsplit` reports differences, adjust the options or types to ensure identical output.

For example, in the example above, pass the `json.FormatNilSliceAsNull(true)` option. In other cases, v2 may report a problem that is straightforward to fix. For example, applying the "string" JSON struct field tag to an invalid type (such as a struct) is ignored in v1, but reports an error in v2. While `ReportErrorsWithLegacySemantics` would suppress the error, it makes more sense to drop the "string" tag. It isn't doing anything anyway.

Note that a difference in output does not necessarily mean that downstream behavior is broken, but that there is an opportunity for breakage. We adjust options now so we can complete the vast majority of the migration without stopping to evaluate subtle output changes, but after switching to v2, you should revisit these locations to determine if you can migrate to the new behavior.

4. Switch to v2.

Once your production environment stops reporting new differences, you can migrate to v2 behavior by setting `OnlyCallV2` or `CallBothButReturnV2` to keep checking for differences.

5. Clean up.

Once the switch to v2 is safely deployed, clean up the migration, by switching from `jsonsplit` to `encoding/json/v2` itself. At this point, you can evaluate the cases from (3) that kept some v1 behavior to determine if it is safe to switch them to the v2 behavior.

See the `jsonsplit` documentation for more details about `jsonsplit` migrations.

□□□ #2
□ □□□ □□□ 2026-10-26 20:26:29 UTC
□ □□□ □□□ 2026-10-26 20:26:29 UTC