# JSON-Patch

`jsonpatch` is a library which provides functionality for both applying RFC6902 JSON patches against documents, as well as for calculating & applying RFC7396 JSON merge patches.


# Get It!

**Latest and greatest**:

```
go get -u github.com/evanphx/json-patch/v5
```

**Stable Versions**:

- Version 5: `go get -u gopkg.in/evanphx/json-patch.v5`
- Version 4: `go get -u gopkg.in/evanphx/json-patch.v4`

(previous versions below `v3` are unavailable)

# Use It!

- Create and apply a merge patch
- Create and apply a JSON Patch
- Comparing JSON documents
- Combine merge patches

# Configuration

- There is a global configuration variable `jsonpatch.SupportNegativeIndices`. This defaults to `true` and enables the non-standard practice of allowing negative indices to mean indices starting at the end of an array. This functionality can be disabled by setting `jsonpatch.SupportNegativeIndices = false`.
- There is a global configuration variable `jsonpatch.AccumulatedCopySizeLimit`, which limits the total size increase in bytes caused by "copy" operations in a patch. It defaults to 0, which means there is no limit.

These global variables control the behavior of `jsonpatch.Apply`.

An alternative to `jsonpatch.Apply` is `jsonpatch.ApplyWithOptions` whose behavior is controlled by an `options` parameter of type `*jsonpatch.ApplyOptions`.

Structure `jsonpatch.ApplyOptions` includes the configuration options above and adds two new options: `AllowMissingPathOnRemove` and `EnsurePathExistsOnAdd`.

When `AllowMissingPathOnRemove` is set to `true`, `jsonpatch.ApplyWithOptions` will ignore `remove` operations whose `path` points to a non-existent location in the JSON document. `AllowMissingPathOnRemove` defaults to `false` which will lead to `jsonpatch.ApplyWithOptions` returning an error when hitting a missing `path` on `remove`.

When `EnsurePathExistsOnAdd` is set to `true`, `jsonpatch.ApplyWithOptions` will make sure that `add` operations produce all the `path` elements that are missing from the target object.

Use `jsonpatch.NewApplyOptions` to create an instance of `jsonpatch.ApplyOptions` whose values are populated from the global configuration variables.

## Create and apply a merge patch

Given both an original JSON document and a modified JSON document, you can create a Merge Patch document.

It can describe the changes needed to convert from the original to the modified JSON document.

Once you have a merge patch, you can apply it to other JSON documents using the `jsonpatch.MergePatch(document, patch)` function.

```
package main

import (
    "fmt"

    jsonpatch "github.com/evanphx/json-patch"
)

func main() {
    // Let's create a merge patch from these two documents...
    original := []byte(`{"name": "John", "age": 24, "height": 3.21}`)
    target := []byte(`{"name": "Jane", "age": 24}`)

    patch, err := jsonpatch.CreateMergePatch(original, target)
    if err != nil {
        panic(err)
    }

    // Now lets apply the patch against a different JSON document...

    alternative := []byte(`{"name": "Tina", "age": 28, "height": 3.75}`)
    modifiedAlternative, err := jsonpatch.MergePatch(alternative, patch)

    fmt.Printf("patch document:   %s\n", patch)
    fmt.Printf("updated alternative doc: %s\n", modifiedAlternative)
}
```

When ran, you get the following output:

```
$ go run main.go
patch document:   {"height":null,"name":"Jane"}
```

```
updated alternative doc: {"age":28,"name":"Jane"}
```

## Create and apply a JSON Patch

You can create patch objects using `DecodePatch([]byte)`, which can then be applied against JSON documents.

The following is an example of creating a patch from two operations, and applying it against a JSON document.

```
package main

import (
    "fmt"

    jsonpatch "github.com/evanphx/json-patch"
)

func main() {
    original := []byte(`{"name": "John", "age": 24, "height": 3.21}`)
    patchJSON := []byte(`[
        {"op": "replace", "path": "/name", "value": "Jane"},
        {"op": "remove", "path": "/height"}
    ]`)

    patch, err := jsonpatch.DecodePatch(patchJSON)
    if err != nil {
        panic(err)
    }

    modified, err := patch.Apply(original)
    if err != nil {
        panic(err)
    }

    fmt.Printf("Original document: %s\n", original)
    fmt.Printf("Modified document: %s\n", modified)
}
```

When ran, you get the following output:

```
$ go run main.go
Original document: {"name": "John", "age": 24, "height": 3.21}
Modified document: {"age":24,"name":"Jane"}
```

## Comparing JSON documents

Due to potential whitespace and ordering differences, one cannot simply compare JSON strings or byte-arrays directly.

As such, you can instead use `jsonpatch.Equal(document1, document2)` to determine if two JSON documents are *structurally* equal. This ignores whitespace differences, and key-value ordering.

```
package main

import (
    "fmt"

    jsonpatch "github.com/evanphx/json-patch"
)

func main() {
    original := []byte(`{"name": "John", "age": 24, "height": 3.21}`)
    similar := []byte(`
        {
            "age": 24,
            "height": 3.21,
            "name": "John"
        }
    `)
    different := []byte(`{"name": "Jane", "age": 20, "height": 3.37}`)

    if jsonpatch.Equal(original, similar) {
        fmt.Println(`"original" is structurally equal to "similar"`)
    }

    if !jsonpatch.Equal(original, different) {
        fmt.Println(`"original" is _not_ structurally equal to "different"`)
    }
}
```

When ran, you get the following output:

```
$ go run main.go
"original" is structurally equal to "similar"
"original" is _not_ structurally equal to "different"
```

## Combine merge patches

Given two JSON merge patch documents, it is possible to combine them into a single merge patch which can describe both set of changes.

The resulting merge patch can be used such that applying it results in a document structurally similar as merging each merge patch to the document in succession.

```
package main

import (
    "fmt"

    jsonpatch "github.com/evanphx/json-patch"
)
```

```
func main() {
    original := []byte(`{"name": "John", "age": 24, "height": 3.21}`)

    nameAndHeight := []byte(`{"height":null,"name":"Jane"}`)
    ageAndEyes := []byte(`{"age":4.23,"eyes":"blue"}`)

    // Let's combine these merge patch documents...
    combinedPatch, err := jsonpatch.MergeMergePatches(nameAndHeight, ageAndEyes)
    if err != nil {
        panic(err)
    }

    // Apply each patch individual against the original document
    withoutCombinedPatch, err := jsonpatch.MergePatch(original, nameAndHeight)
    if err != nil {
        panic(err)
    }

    withoutCombinedPatch, err = jsonpatch.MergePatch(withoutCombinedPatch,
ageAndEyes)
    if err != nil {
        panic(err)
    }

    // Apply the combined patch against the original document

    withCombinedPatch, err := jsonpatch.MergePatch(original, combinedPatch)
    if err != nil {
        panic(err)
    }

    // Do both result in the same thing? They should!
    if jsonpatch.Equal(withCombinedPatch, withoutCombinedPatch) {
        fmt.Println("Both JSON documents are structurally the same!")
    }

    fmt.Printf("combined merge patch: %s", combinedPatch)
}
```

When ran, you get the following output:

```
$ go run main.go
Both JSON documents are structurally the same!
combined merge patch: {"age":4.23,"eyes":"blue","height":null,"name":"Jane"}
```

# CLI for comparing JSON documents

You can install the commandline program `json-patch`.

This program can take multiple JSON patch documents as arguments, and fed a JSON document from `stdin`. It will apply the patch(es) against the document and output the modified doc.

**patch.1.json**

```
[
    {"op": "replace", "path": "/name", "value": "Jane"},
    {"op": "remove", "path": "/height"}
]
```

**patch.2.json**

```
[
    {"op": "add", "path": "/address", "value": "123 Main St"},
    {"op": "replace", "path": "/age", "value": "21"}
]
```

**document.json**

```
{
    "name": "John",
    "age": 24,
    "height": 3.21
}
```

You can then run:

```
$ go install github.com/evanphx/json-patch/cmd/json-patch
$ cat document.json | json-patch -p patch.1.json -p patch.2.json
{"address":"123 Main St","age":"21","name":"Jane"}
```

# Help It!

Contributions are welcomed! Leave an issue or create a PR.

Before creating a pull request, we'd ask that you make sure tests are passing and that you have added new tests when applicable.

Contributors can run tests using:

```
go test -cover ./...
```

Builds for pull requests are tested automatically using TravisCI.