

[Discover Packages](#) > [Standard library](#) > [embed](#) 

embed


package

standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: 4 |

Imported by: 16,270

Details

 Valid [go.mod](#) file  Redistributable license  Tagged version  Stable version [Learn more](#)

Repository

[cs.opensource.google/go/go](#)

Links

 [Report a Vulnerability](#) Documentation 

<> Documentation

Overview

[Directives](#)[Strings and Bytes](#)[File Systems](#)[Tools](#)

Package embed provides access to files embedded in the running Go program.

Go source files that import "embed" can use the `//go:embed` directive to initialize a variable of type string, `[]byte`, or `FS` with the contents of files read from the package directory or subdirectories at compile time.

For example, here are three ways to embed a file named `hello.txt` and then print its contents at run time.

Embedding one file into a string:

```
import _ "embed"

//go:embed hello.txt
var s string
print(s)
```

Embedding one file into a slice of bytes:

```
import _ "embed"

//go:embed hello.txt
```

```
var b []byte
print(string(b))
```

Embedded one or more files into a file system:

```
import "embed"

//go:embed hello.txt
var f embed.FS
data, _ := f.ReadFile("hello.txt")
print(string(data))
```

Directives

A `//go:embed` directive above a variable declaration specifies which files to embed, using one or more path.Match patterns.

The directive must immediately precede a line containing the declaration of a single variable. Only blank lines and `//` line comments are permitted between the directive and the declaration.

The type of the variable must be a string type, or a slice of a byte type, or FS (or an alias of FS).

For example:

```
package server

import "embed"

// content holds our static web server content.
//go:embed image/* template/*
//go:embed html/index.html
var content embed.FS
```

The Go build system will recognize the directives and arrange for the declared variable (in the example above, `content`) to be populated with the matching files from the file system.

The `//go:embed` directive accepts multiple space-separated patterns for brevity, but it can also be repeated, to avoid very long lines when there are many patterns. The patterns are interpreted relative to the package directory containing the source file. The path separator is a forward slash, even on Windows systems. Patterns may not contain `'.'` or `'..'` or empty path elements, nor may they begin or end with a slash. To match everything in the current directory, use `'*'` instead of `'.'`. To allow for naming files with spaces in their names, patterns can be written as Go double-quoted or back-quoted string literals.

If a pattern names a directory, all files in the subtree rooted at that directory are embedded (recursively), except that files with names beginning with `'.'` or `'_'` are excluded. So the variable in the above example is almost equivalent to:

```
// content is our static web server content.
//go:embed image template html/index.html
```

```
var content embed.FS
```

The difference is that `'image/*'` embeds `'image/.tempfile'` while `'image'` does not. Neither embeds `'image/dir/.tempfile'`.

If a pattern begins with the prefix `'all:.'`, then the rule for walking directories is changed to include those files beginning with `'.'` or `'_'`. For example, `'all:image'` embeds both `'image/.tempfile'` and `'image/dir/.tempfile'`.

The `//go:embed` directive can be used with both exported and unexported variables, depending on whether the package wants to make the data available to other packages. It can only be used with variables at package scope, not with local variables.

Patterns must not match files outside the package's module, such as `'git/*'` or symbolic links. Patterns must not match files whose names include the special punctuation characters `" * < > ? ` ' | / \` and `..`. Matches for empty directories are ignored. After that, each pattern in a `//go:embed` line must match at least one file or non-empty directory.

If any patterns are invalid or have invalid matches, the build will fail.

Strings and Bytes

The `//go:embed` line for a variable of type `string` or `[]byte` can have only a single pattern, and that pattern can match only a single file. The `string` or `[]byte` is initialized with the contents of that file.

The `//go:embed` directive requires importing `"embed"`, even when using a `string` or `[]byte`. In source files that don't refer to `embed.FS`, use a blank import (`import _ "embed"`).

File Systems

For embedding a single file, a variable of type `string` or `[]byte` is often best. The `FS` type enables embedding a tree of files, such as a directory of static web server content, as in the example above.

`FS` implements the `io/fs` package's `FS` interface, so it can be used with any package that understands file systems, including `net/http`, `text/template`, and `html/template`.

For example, given the `content` variable in the example above, we can write:

```
http.Handle("/static/", http.StripPrefix("/static/", http.FileServer(http.FS(content))))

template.ParseFS(content, "*.tmpl")
```

Tools

To support tools that analyze Go packages, the patterns found in `//go:embed` lines are available in “go list” output. See the `EmbedPatterns`, `TestEmbedPatterns`, and `XTestEmbedPatterns` fields in the “go help list” output.

► [Example](#)

Index

type FS

```
func (f FS) Open(name string) (fs.File, error)
func (f FS) ReadDir(name string) ([]fs.DirEntry, error)
func (f FS) ReadFile(name string) ([]byte, error)
```

Examples

Package

Constants

This section is empty.

Variables

This section is empty.

Functions

This section is empty.

Types

type FS

```
type FS struct {
    // contains filtered or unexported fields
}
```

An FS is a read-only collection of files, usually initialized with a `//go:embed` directive. When declared without a `//go:embed` directive, an FS is an empty file system.

An FS is a read-only value, so it is safe to use from multiple goroutines simultaneously and also safe to assign values of type FS to each other.

FS implements `fs.FS`, so it can be used with any package that understands file system interfaces, including `net/http`, `text/template`, and `html/template`.

See the package documentation for more details about initializing an FS.

func (FS) Open

```
func (f FS) Open(name string) (fs.File, error)
```

Open opens the named file for reading and returns it as an `fs.File`.

The returned file implements `io.Seeker` when the file is not a directory.

func (FS) ReadDir

```
func (f FS) ReadDir(name string) ([]fs.DirEntry, error)
```

ReadDir reads and returns the entire named directory.

func (FS) ReadFile

```
func (f FS) ReadFile(name string) ([]byte, error)
```

ReadFile reads and returns the content of the named file.



Source Files

[View all](#) 

[embed.go](#)

Why Go

[Use Cases](#)

[Case Studies](#)

Get Started

[Playground](#)

[Tour](#)

[Stack Overflow](#)

[Help](#)

Packages

[Standard Library](#)

[About Go Packages](#)

About

[Download](#)

[Blog](#)

[Issue Tracker](#)

[Release Notes](#)

[Brand Guidelines](#)

[Code of Conduct](#)

Connect

[Twitter](#)

[GitHub](#)

[Slack](#)

[r/golang](#)

[Meetup](#)

[Golang Weekly](#)

Copyright

[Terms of Service](#)

[Privacy Policy](#)

[Report an Issue](#)

