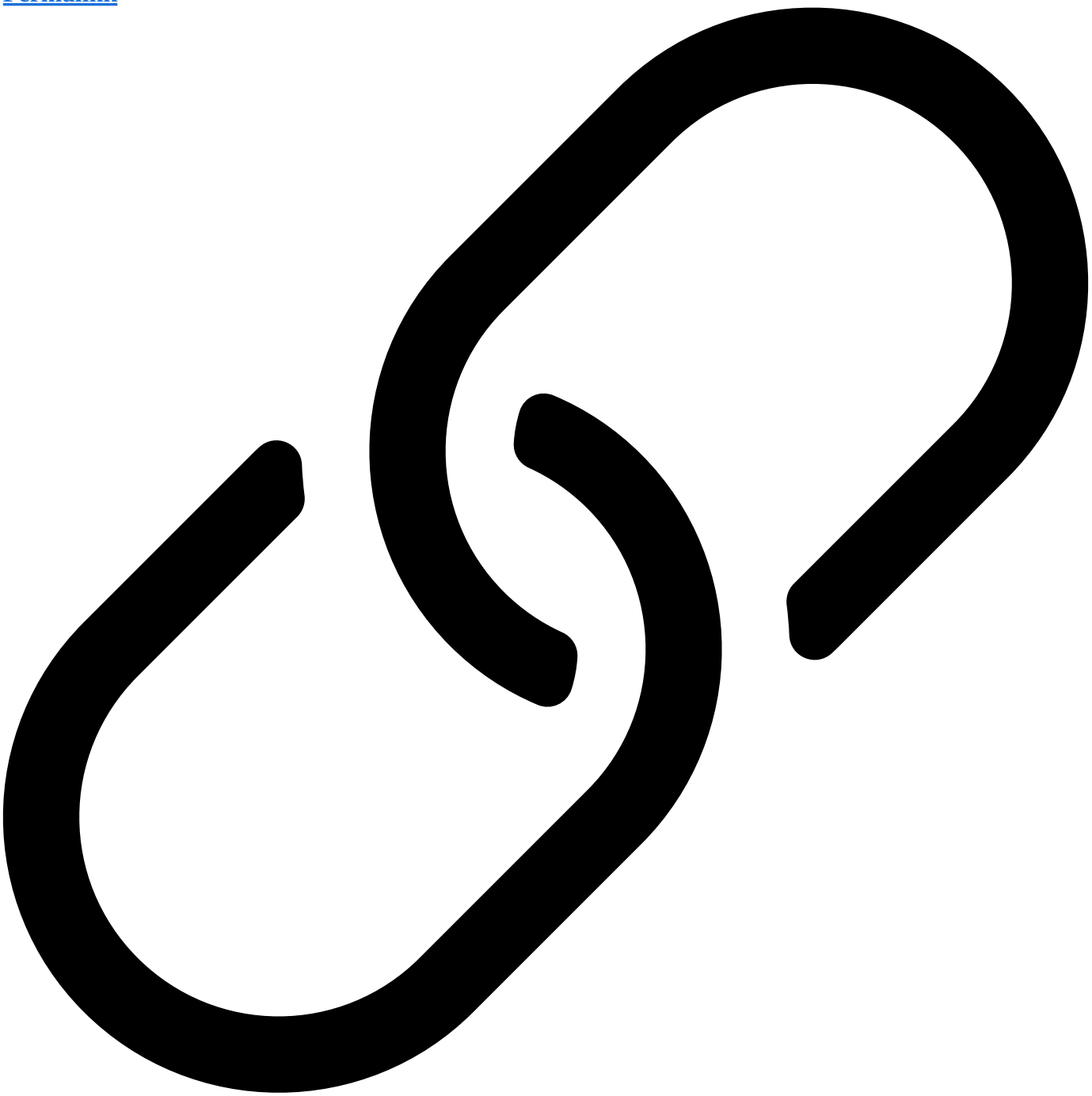
A Guide to Effective Go Documentation

Nirdosh Gautam

In this post, we'll explore Go documentation, tools and techniques to make it effective, and ways to share it.

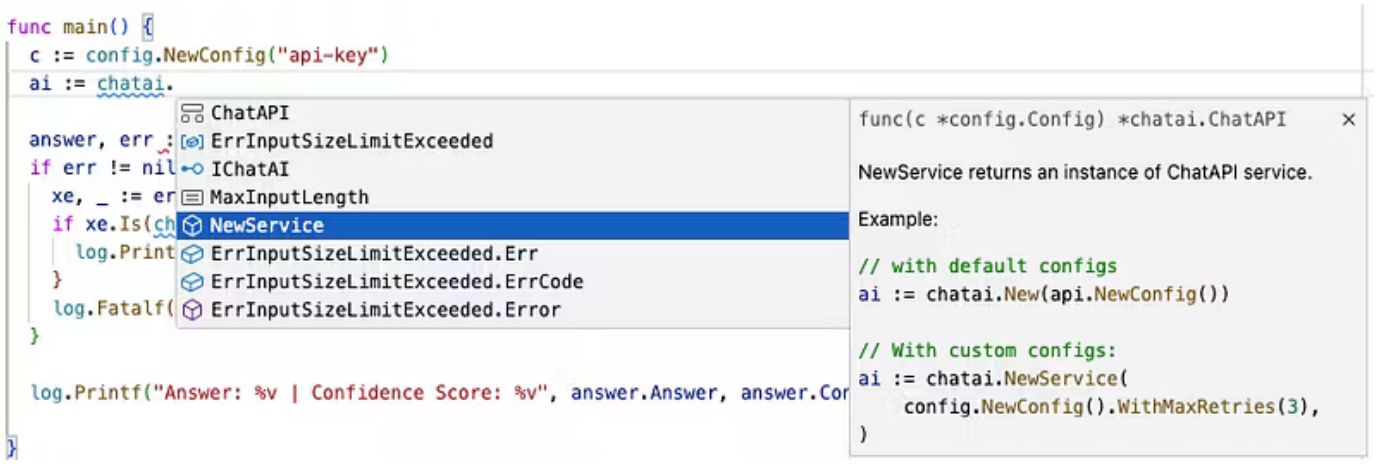
Permalink



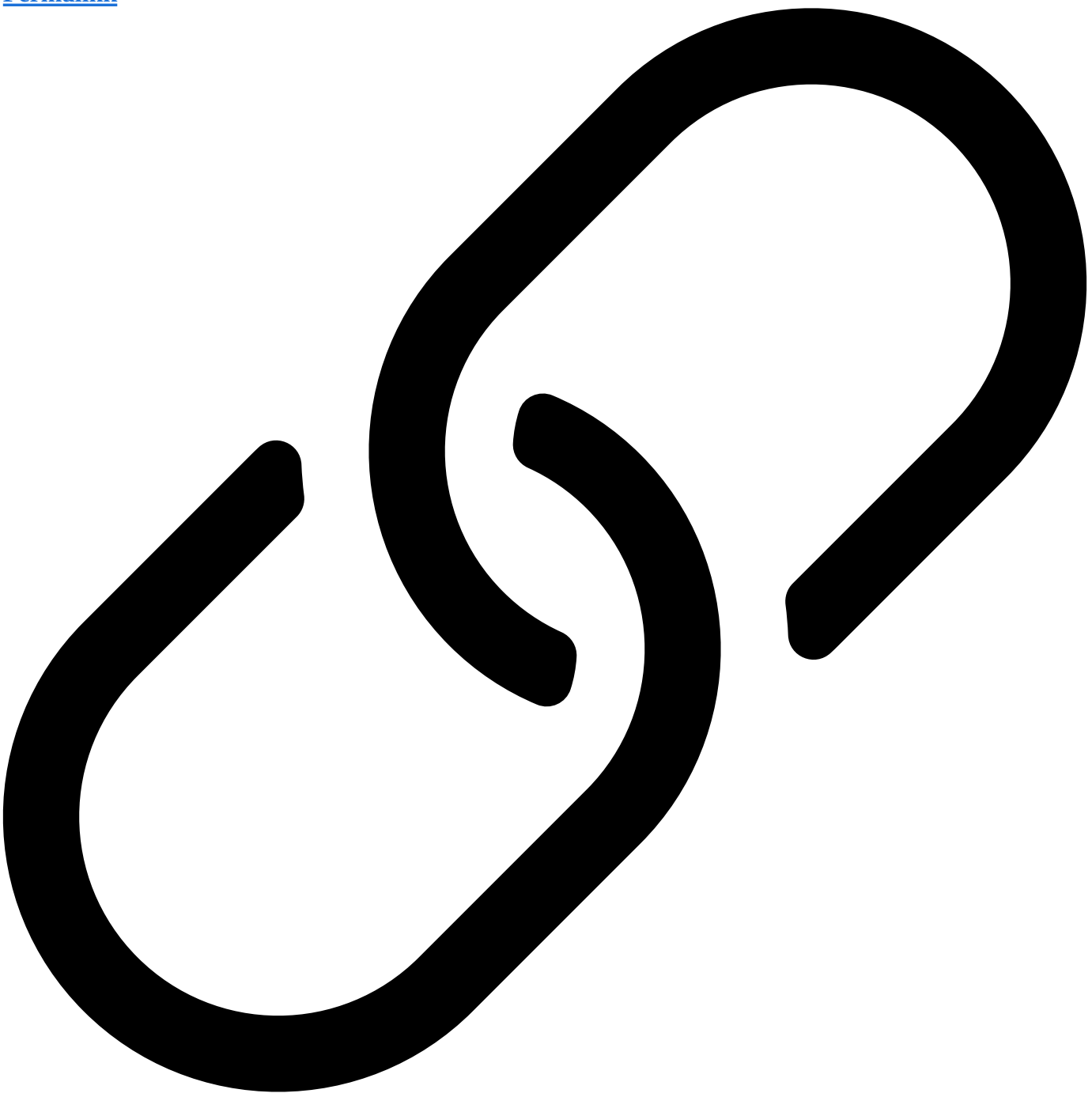
What should be documented?

Any exported type, function, constant, and variable become **visible** for public use, so they should be well documented. Other packages inside the same library or entirely different project can use the functionalities provided by the exported package.

Code editors also parse the comments in Go code and present documentation while we are writing code. So, it's up to you to decide what information to share with the developers so that it becomes easy to understand and use your code.



[Permalink]

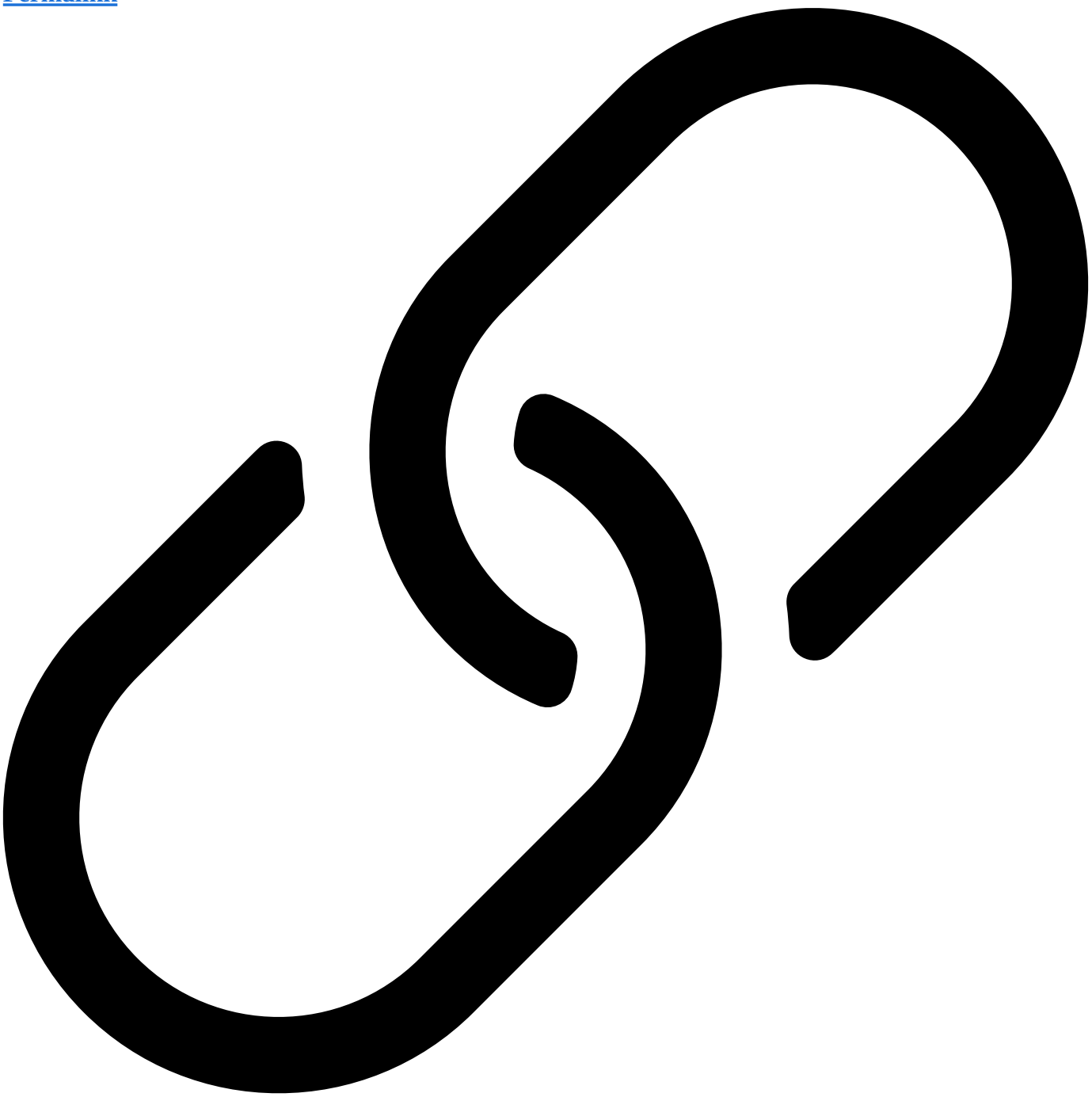


pkg.go.dev

pkg.go.dev keeps an index of public libraries and hosts documentation for them. Pubic packages are indexed automatically when someone performs `go get` or we can also manually add or remove packages.

Pkg site is not for private libraries, so we have to host our documentation somewhere else.

Permalink



Godoc tool

Godoc is a tool that parses Go source code with comments to generate human-readable documentation in an HTML site. Whatever documentation you see in pkg.go.dev is generated by Godoc.

Only exported types, variables, constants, and functions are visible in the documentation. This is because unexported types, which are not intended to be used by external code, are excluded.

Installing Godoc:

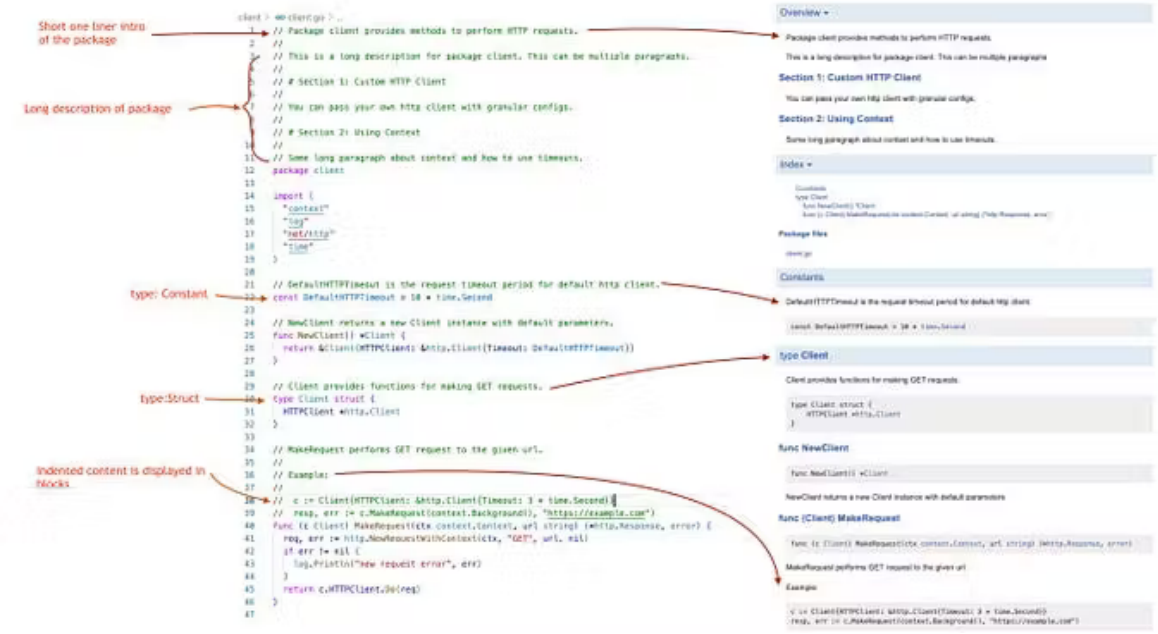
```
go install golang.org/x/tools/cmd/godoc@latest
```

Viewing documentation locally:

```
godoc -http=:8080
```

There are some conventions that we follow in writing comments that allow parsers like Godoc to identify different sections of documentation such as short intro text, detailed descriptions, examples, etc.

Let’s look into the comments in this sample code and what the generated documentation looks like:



You can see the following information generated from the comments:

- **The title or short description**
  should be the first comment for the package without any new lines between the comment and the package definition.

- **Long description**
  Starts after topic sentences after a new line. Can contain as many paragraphs and sections as we want.

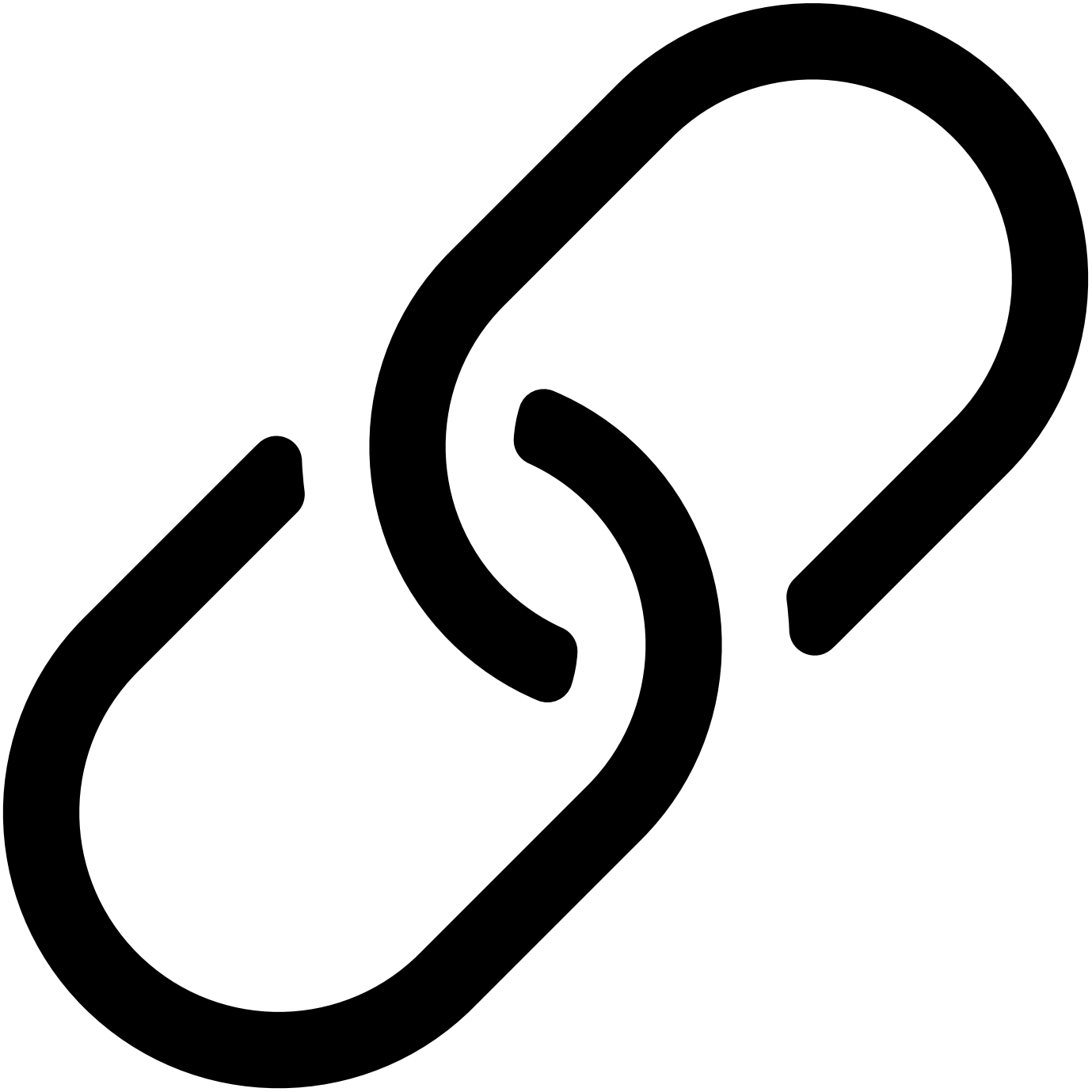
- **Sub-sections/headings**
  Starts with “#” and is displayed as a sub-heading

- **Code Examples**
  If we indent anything in the description section, it is displayed inside a block, whether it is code or not. There is no syntax highlighting for code in documentation generated by godoc, but code editors do highlight the code intelligently as shown below:

```
// Client provides functions for making GET requests.
type Client struct {
  HTTPClient *http.Client
}

// MakeRequest p
//
// Example:
//
//  c := Client{
//  resp, err :=
func (c Client) MakeRequest(ctx context.Context, url string) (*http.Response, error) {
  req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
  if err != nil {
    log.Println("new request error", err)
  }
  return c.HTTPClient.Do(req)
}
```

```
func (Client).MakeRequest(ctx context.Context, url string) (*http.Response, error)

MakeRequest performs GET request to the given url.

Example:

c := Client{HTTPClient: &http.Client{Timeout: 3 * time.Second}}
resp, err := c.MakeRequest(context.Background(), "https://example.com")

(client.Client).MakeRequest on pkg.go.dev
```

Permalink



Testable Examples

As the name suggests, Testable Examples in Go are pieces of code that act as both documentation as well as tests.

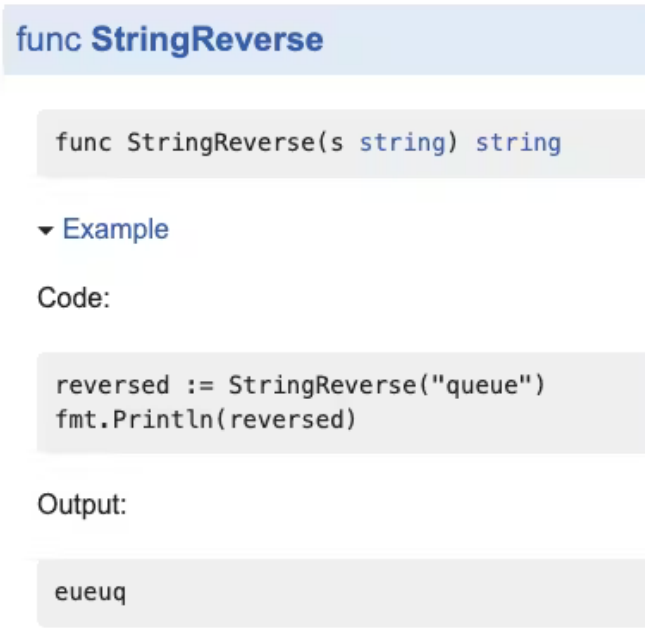
They are compiled and run by `go test` and also displayed in the package documentation although they are not comments.

File naming convention: `example_test.go` or `example_{FUNCTIONALITY_NAME}_test.go`

Super important: Similar to test functions which start with the word `Test`, example functions start with the word `Example` but they don't take any parameters.

```
package mypkg

import "fmt"

func ExampleStringReverse() {
    reversed := StringReverse("queue")
    fmt.Println(reversed)
    // Output: eueuq
}
```

If you run `godoc`, documentation for the above code will look like this:

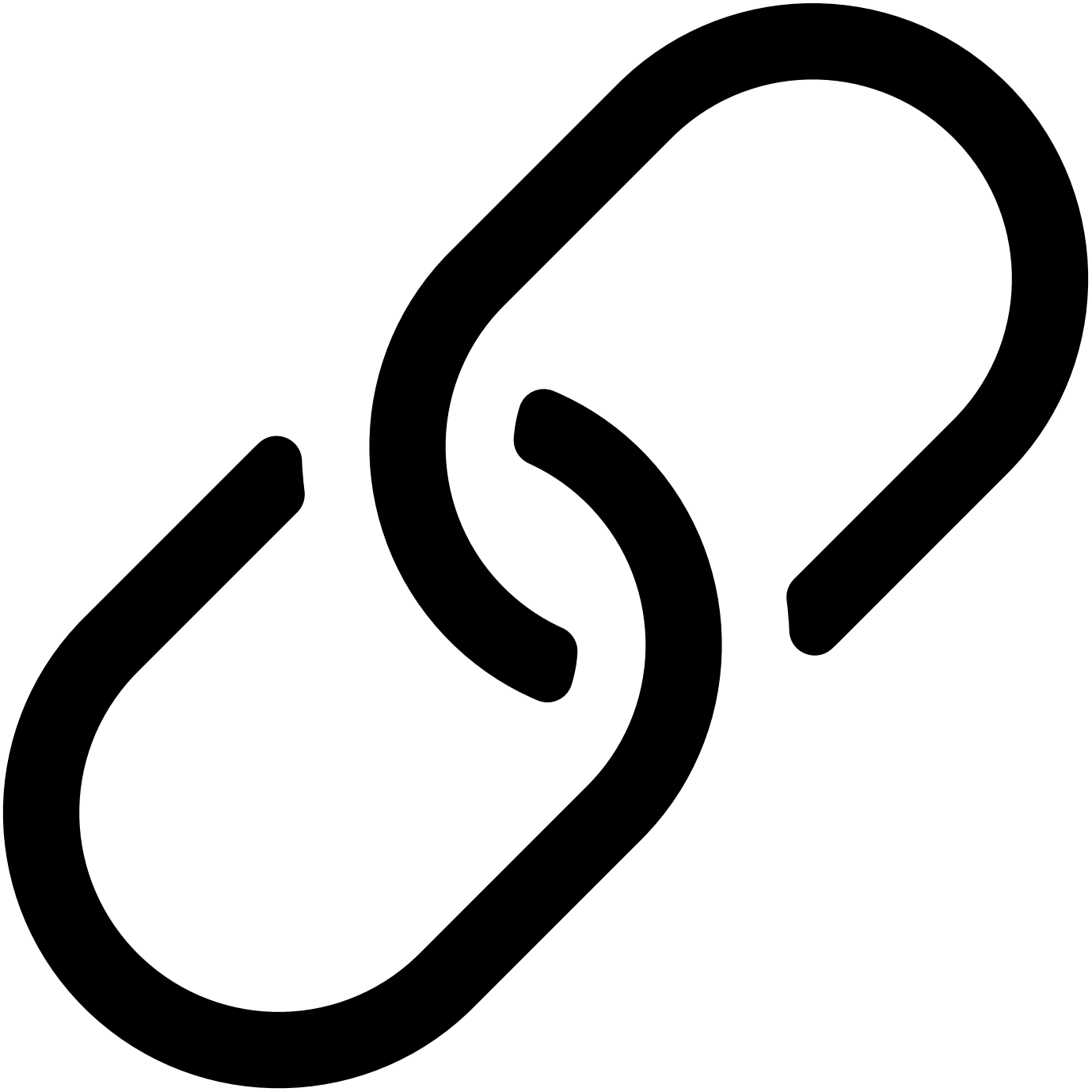


Output Comment:

If the output comment `// Output:` is present, it captures the last printed output and compares it with the expected value which is defined after `// Output:`. If the output matches, the test passes else fails.

Know more about testable examples.

Permalink



Manage lengthy documentation using doc.go

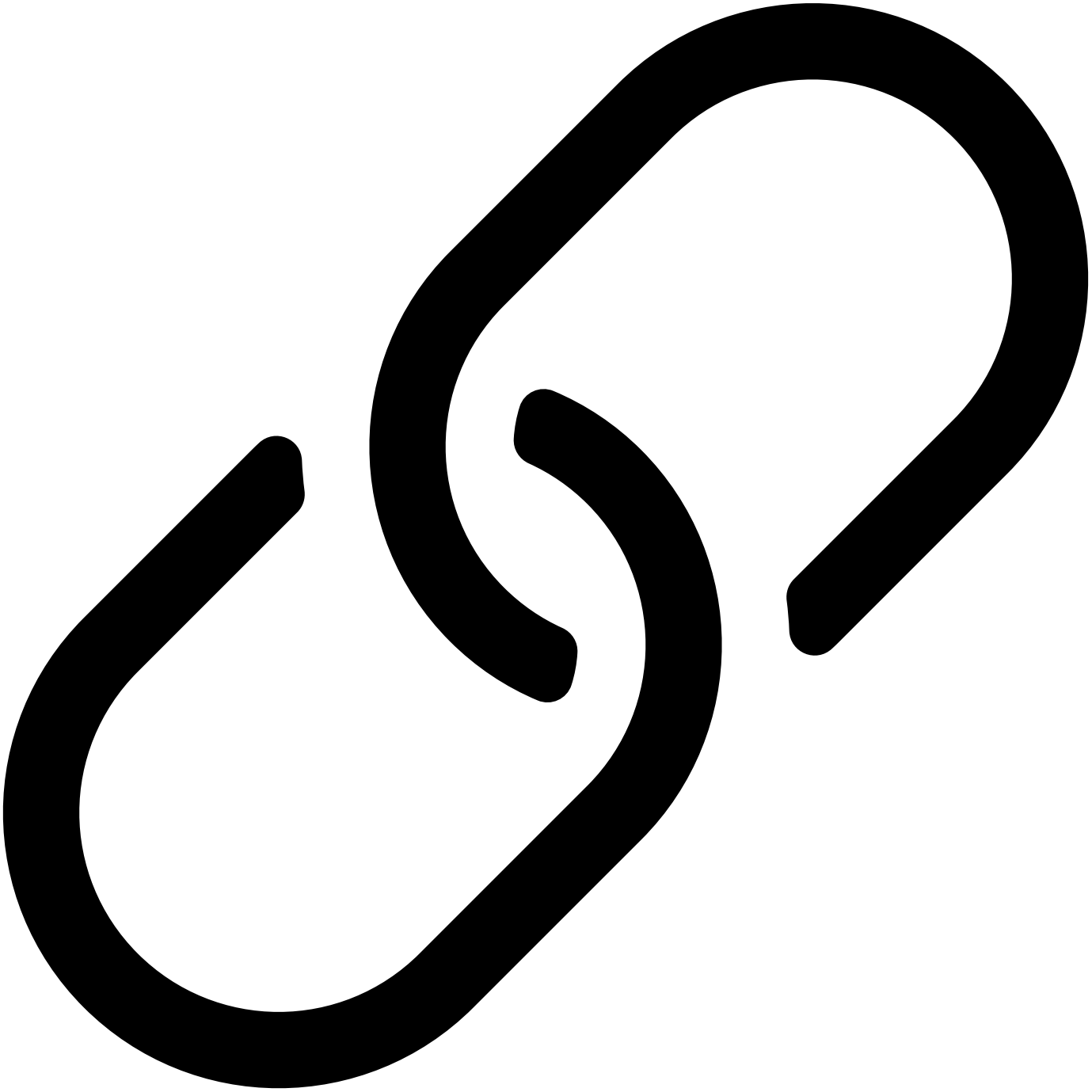
If your package-level introductory documentation is very long, you can create a `doc.go` file. It only contains comments that are written following Godoc conventions.

For example, *this doc.go file from aws-sdk-go* is displayed like *THIS*.

We can also add a *doc.go* file inside sub-packages if necessary.

```
my-sdk
├── client
│   └── client.go
│   └── doc.go # documentation for subpackage 'client'
├── doc.go   # documentation for the whole package 'my-sdk'
└── go.mod
```

Permalink



Generating and publishing documentation

We have written good documentation for our library. Now what?

Now you can view documentation locally and publish it to public indexes like Go pkg site where everyone can access or manage the docs privately as described below:

Generate and view documentation from Go source code locally:

- Run godoc server in root dir of your source code : `godoc -http=:8080`

Host documentation for public access:

- If your package is **public,** it will be indexed automatically in pkg.go.dev once you perform `go get` or you can add it manually.

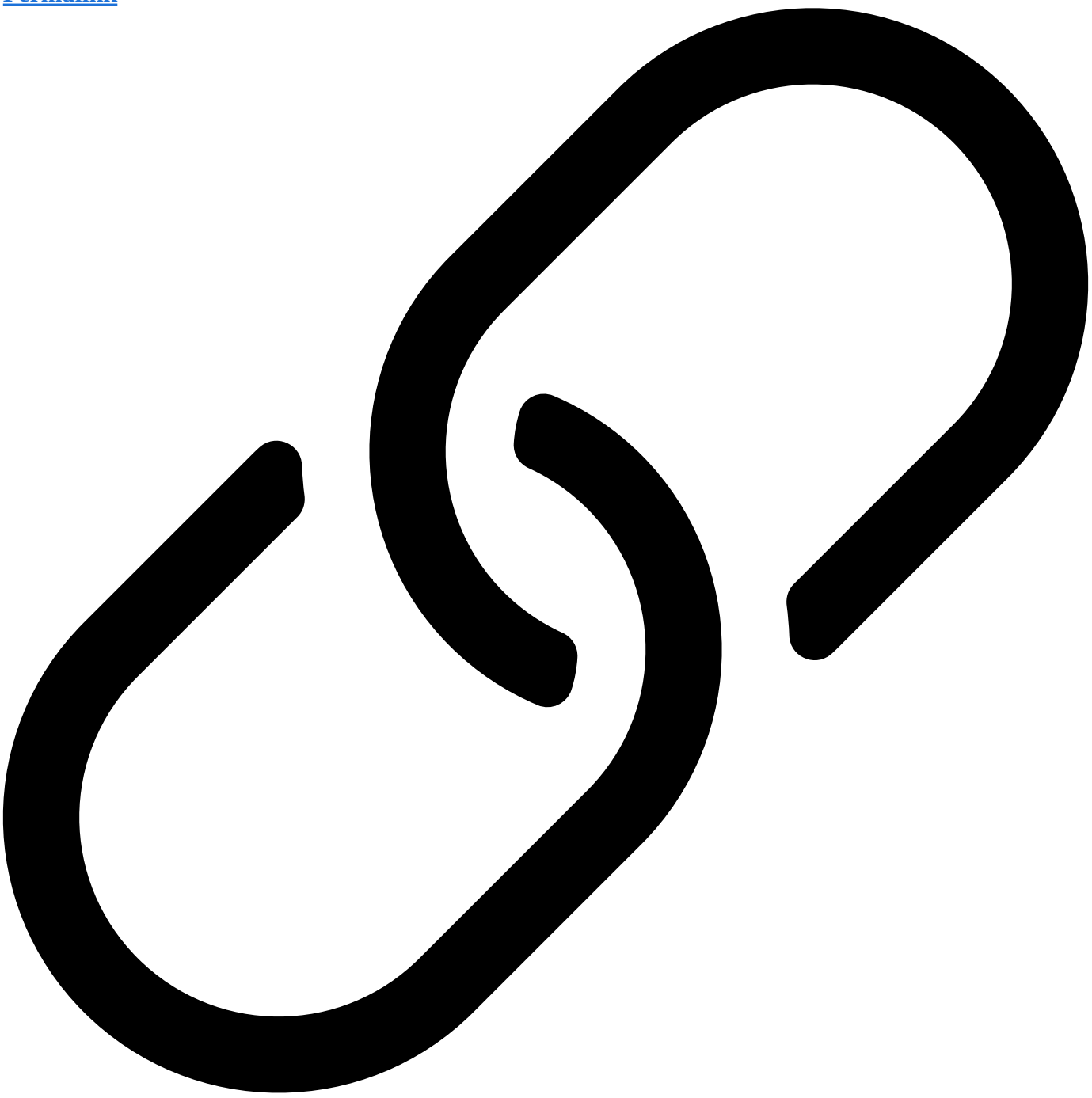
Host documentation for private access:

- Start Godoc server in any public or private instance you like and you can add an authentication layer on top of it.
- Start Godoc, copy all the generated HTML/JS, and CSS files, and host them as a static site e.g. Amazon S3. Godoc natively does not expose generated static files, so we have to apply a trick to download the assets as shown

below:

```
$ godoc -http=:8080 &
wget -r -np -N -E -p -k http://localhost:8080/pkg/{your_package_link}
```

Permalink



Can we generate a zero-dependency HTML file?

In API specs like Swagger/OpenAPI, we can convert the API specs YAML file directly to a single standalone HTML file using tools like Redoc. It is a lot simpler to automate doc generation and host a single static file.

Unfortunately, there is no such feature offered by `godoc` and there is no straightforward way to do this.

Here are some workarounds:

Option 1:

- Create a markdown(e.g. Github Readme) and document the most useful part of your library(public types). You can copy and paste from the documentation generated by Godoc.
- Then, use static site generators like Hugo to generate a static site from markdown and serve your doc.

Option 2:

- It is a hectic job but assets(CSS, js, images) copied from Godoc using wget can be manually injected into the main HTML file by adding <script>, <style>, and <img> tags.
- Minify JS and CSS, convert images to base64 and add them inline in HTML.