Our User Interface

This lesson shows the implementation of the application by designing a server and making a front end to see how things work.

WE'LL COVER THE FOLLOWING

Designing a server

We haven't yet coded the function with which our program must be started. This is (always) the function `main()` as in C, C++ or Java. In it, we will start our web server, e.g., we can start a local web server on port **8080** with the command:

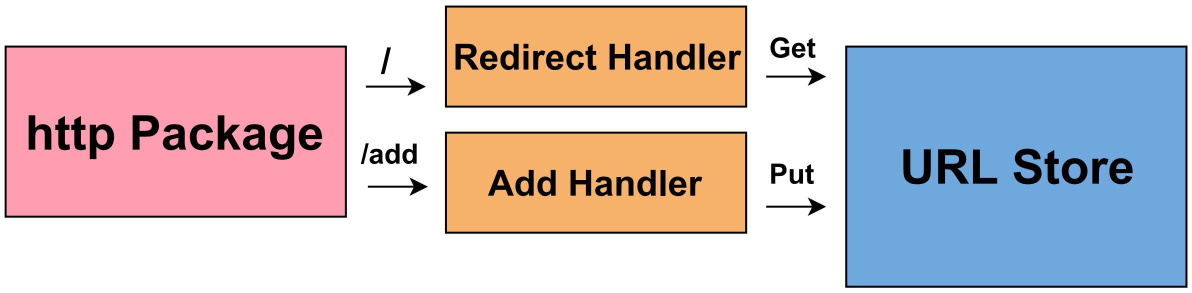
```
http.ListenAndServe(":8080", nil)
```

The web server listens for incoming requests in an infinite loop, but we must also define how this server responds to these requests. We do this by making so-called HTTP handlers with the function `HandleFunc`. For example, by coding `http.HandleFunc("/add", Add)` we say that every request which ends in `/add` will call a function `Add` (still to be made).

Our program will have two HTTP handlers:

- `Redirect`, which redirects short URL requests
- `Add`, which handles the submission of new URLs

Schematically:



Handler Functions in goto

Our minimal `main()` could look like:

```go
func main() {
  http.HandleFunc("/", Redirect)
  http.HandleFunc("/add", Add)
  http.ListenAndServe(":8080", nil)
}
```

Requests to `/add` will be served by the `Add` handler, where all the other requests will be served by the `Redirect` handler. Handler functions get information from an incoming request (a variable r of type `*http.Request`), and they make and write their response to a variable `w` of type `http.ResponseWriter`.

The `Add` function

What must our `Add` function do?

- Read in the long URL, which means reading the URL from an HTML-form contained in an HTTP request with `r.FormValue("url")`.
- Put it in the store using our `Put` method on the store.
- Send the corresponding short URL to the user.

Each requirement translates in one code line:

```go
func Add(w http.ResponseWriter, r *http.Request) {
  url := r.FormValue("url")
  key := store.Put(url)

  fmt.Fprintf(w, "%s", key)
}
```

The function `Fprintf` of the `fmt` package is used here to substitute a key in the string containing %s and then sends that string as a response back to the client. Notice that `Fprintf` writes to a `ResponseWriter`. In fact, `Fprintf` can write to any data structure that implements `io.Writer()`, which means that it implements a `Write()` method. `io.Writer()` is what is called, in Go, an **interface**. We see that through the use of interfaces, `Fprintf` is very general; it can write to a lot of different things. The use of interfaces is pervasive in Go and makes code more generally applicable. However, we still need a form; we can display a form by using `Fprintf` again. This time writing a constant to `w`. Let's modify Add to display an HTML form when no URL is supplied:

```
func Add(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/html")
  url := r.FormValue("url")
  if url == "" {
    fmt.Fprint(w, AddForm)
    return
  }
  key := store.Put(url)

  fmt.Fprintf(w, "%s", key)
}
const AddForm = `
<html><body>
<form method="POST" action="/add">
URL: <input type="text" name="url">
<input type="submit" value="Add">
</form>
<\html><\body>`
```

In that case, we send the constant string `AddForm` to the client, which is the html, necessary for creating a form with an input field `URL`, and a submit button, which when pushed will post a request ending in `/add`. So, the `Add` handler function is again invoked, now with a value for URL from the text field. (The `` are needed to make a raw string, otherwise, strings are enclosed in "" as usual).

The `Redirect` function

The `Redirect` function finds the key in the HTTP request path (the short URL

The `Redirect` function finds the key in the HTTP request path (the short URL key is the request path minus the first character, this can be written in Go as

[1:]; for the request “/abc” the key would be “abc”), retrieves the corresponding long URL from the store with the `Get` function, and sends an HTTP redirect to the user. If the URL is not found, a 404 “Not Found” error is sent instead.

```go
func Redirect(w http.ResponseWriter, r *http.Request) {
  key := r.URL.Path[1:]
  url := store.Get(key)
  if url == "" {
    http.NotFound(w, r)
    return
  }
  http.Redirect(w, r, url, http.StatusFound)
}
```

Here, `http.NotFound` and `http.Redirect` are helpers for sending common HTTP responses.

Now, we have discussed all the code of Version-1. Compile and run the program, which starts the web-server:

Environment Variables

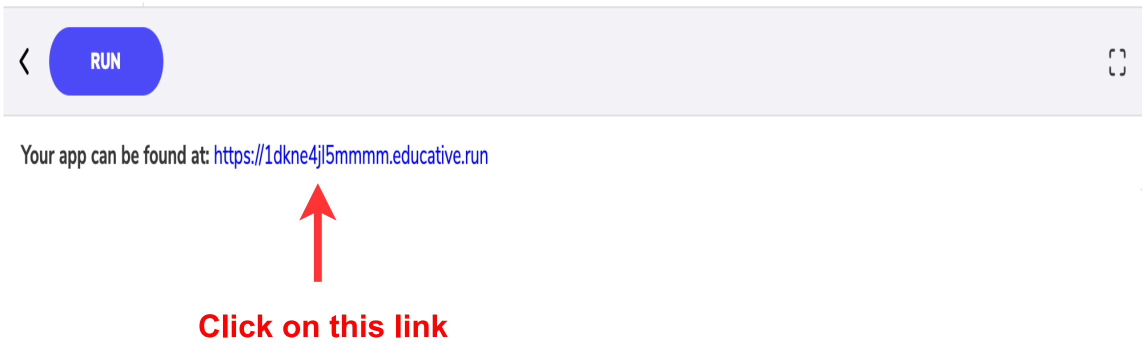
Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```go
package main

import "sync"

type URLStore struct {
        urls  map[string]string
        mu    sync.RWMutex
}

func NewURLStore() *URLStore {
        return &URLStore{urls: make(map[string]string)}
}

func (s *URLStore) Get(key string) string {
        s.mu.RLock()
        defer s.mu.RUnlock()
        return s.urls[key]
}

func (s *URLStore) Set(key, url string) bool {

        s.mu.Lock()
        defer s.mu.Unlock()
        if _, present := s.urls[key]; present {
                return false
        }
        s.urls[key] = url
        return true
}

func (s *URLStore) Count() int {
        s.mu.RLock()
        defer s.mu.RUnlock()
        return len(s.urls)
}

func (s *URLStore) Put(url string) string {
        for {
                key := genKey(s.Count()) // generate the short URL
                if ok := s.Set(key, url); ok {
                        return key
                }
        }
        // shouldn't get here
        return ""
}
```

> **Note**: If you're running this code on your *local machine*, use port number `8080` instead of `3000` at **line 21**.

Testing the program

Click the **Run** button, and wait for the terminal to start. Once it starts, type `go run *.go` and perform the following steps:

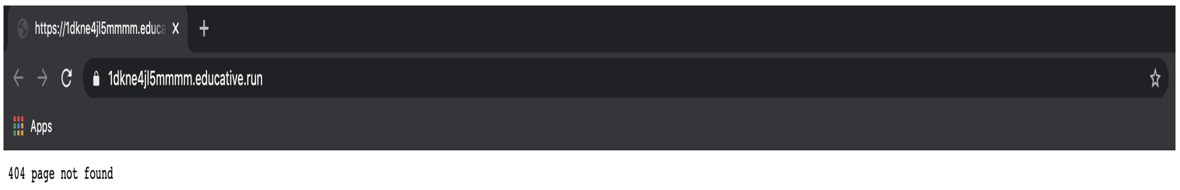


Your app can be found at: https://1dkne4jl5mmmm.educative.run

Step 1

1 of 3

Browser opens a page

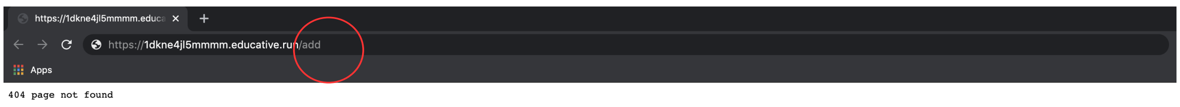


```
404 page not found
```

Step 1

2 of 3

Add "/add" at the end of the URL

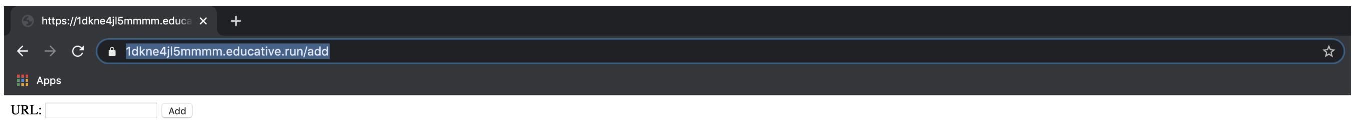

https://1dkne4jl5mmmm.educa
https://1dkne4jl5mmmm.educative.run/add
Apps

```
404 page not found
```

Step 2

3 of 3

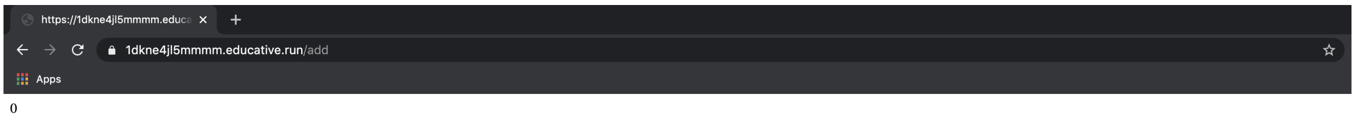
This starts our `Add` handler function. There isn't yet any `url` variable in the form, so the response is the HTML-form which asks for input:



URL:

Add Handler

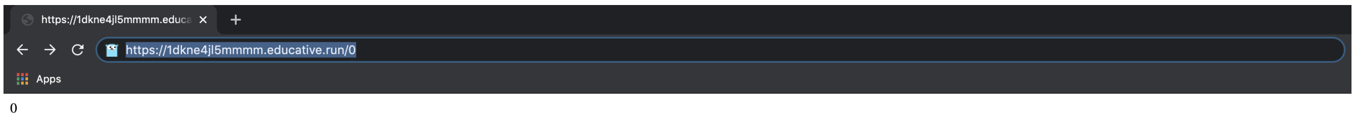
Add a (long) URL for which you want a short equivalent, like http://golang.org/pkg/bufio/#Writer, and press the button. The application makes a short URL for you and prints a key, e.g., 0.



```
0
```

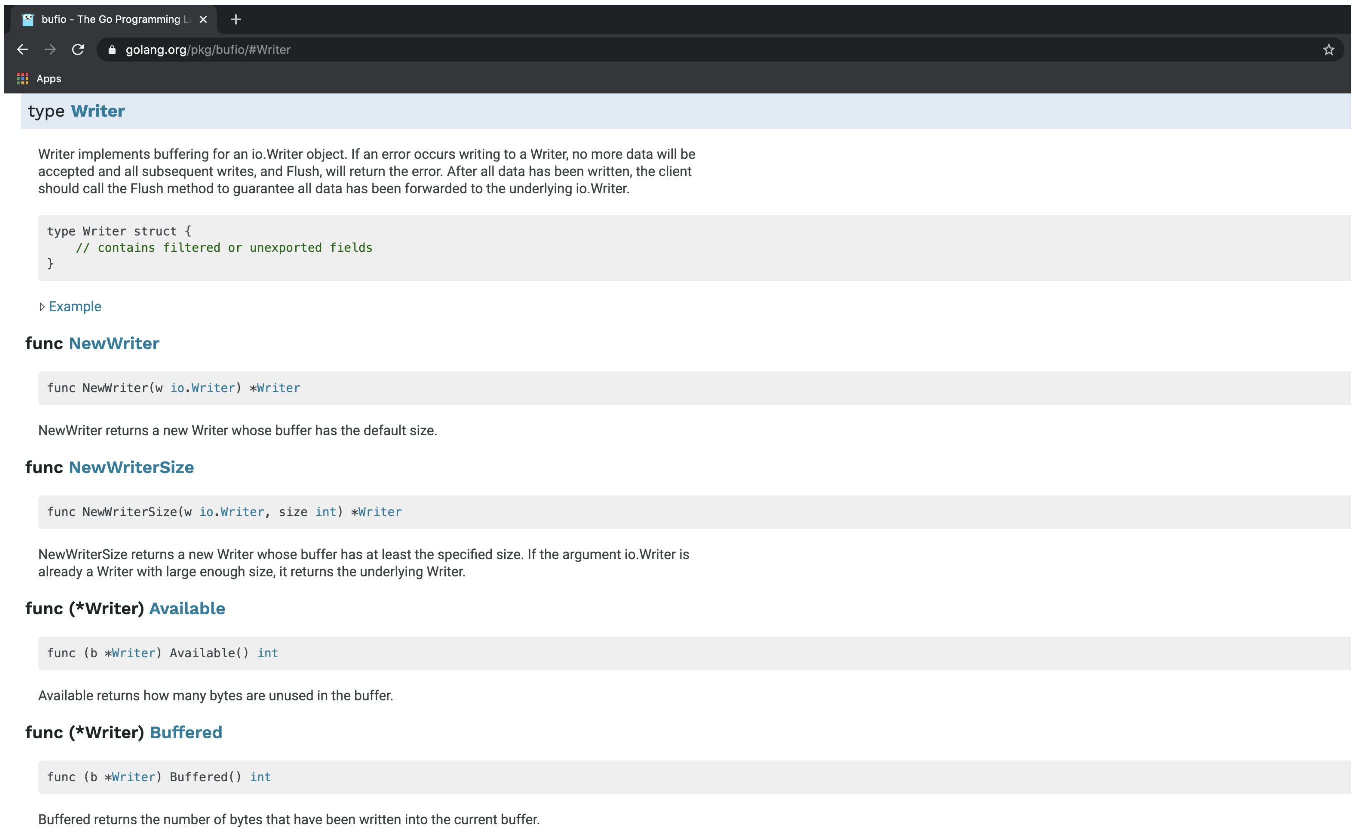
Response of Add Handler

Copy and paste the same URL you did before in your browser address box and append the key at the end to it, received as a response by `Add` handler.


https://1dkne4jl5mmmm.educa
https://1dkne4jl5mmmm.educative.run/0
Apps

```
0
```

Redirect Handler



type **Writer**

Writer implements buffering for an io.Writer object. If an error occurs writing to a Writer, no more data will be accepted and all subsequent writes, and Flush, will return the error. After all data has been written, the client should call the Flush method to guarantee all data has been forwarded to the underlying io.Writer.

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

Example

func NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter returns a new Writer whose buffer has the default size.

func NewWriterSize

```
func NewWriterSize(w io.Writer, size int) *Writer
```

NewWriterSize returns a new Writer whose buffer has at least the specified size. If the argument io.Writer is already a Writer with large enough size, it returns the underlying Writer.

func (*Writer) Available

```
func (b *Writer) Available() int
```

Available returns how many bytes are unused in the buffer.

func (*Writer) Buffered

```
func (b *Writer) Buffered() int
```

Buffered returns the number of bytes that have been written into the current buffer.

Response of Redirect Handler

The result is the `Redirect` handler in action, and the page of the long URL is shown. You can stop this process with CTRL/C in the terminal window.

> **Note**: If you are running this application on a *local machine*, to add a URL, open a browser and request the URL: http://localhost:8080/add. Then, add the long URL. Let’s suppose the server responds with 0 as a key. Lastly, request the URL: http://localhost:8080/0.

Our application is now successful in redirecting to the same page with the shorter URL. Our next concern is having proper storage for URLs that have already been made shorter. Let’s cover this in detail in the next lesson.