

## CONTENTS

Prerequisites

Setting Up the Project

Listening for Requests and Serving Responses

Multiplexing Request Handlers

Running Multiple Servers at One Time

Inspecting a Request's Query String

Reading a Request Body

Retrieving Form Data

Responding with Headers and a Status Code

Conclusion

## RELATED

CodeIgniter: Getting Started With a Simple Example

[View](#) 

How To Install Go and Revel on an Ubuntu 13.04 x64 VPS

[View](#) 

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

MANAGE CHOICES

AGREE & PROCEED



50/53 How To Make an HTTP S...

51/53 How To Make HTTP Requ...



// Tutorial //

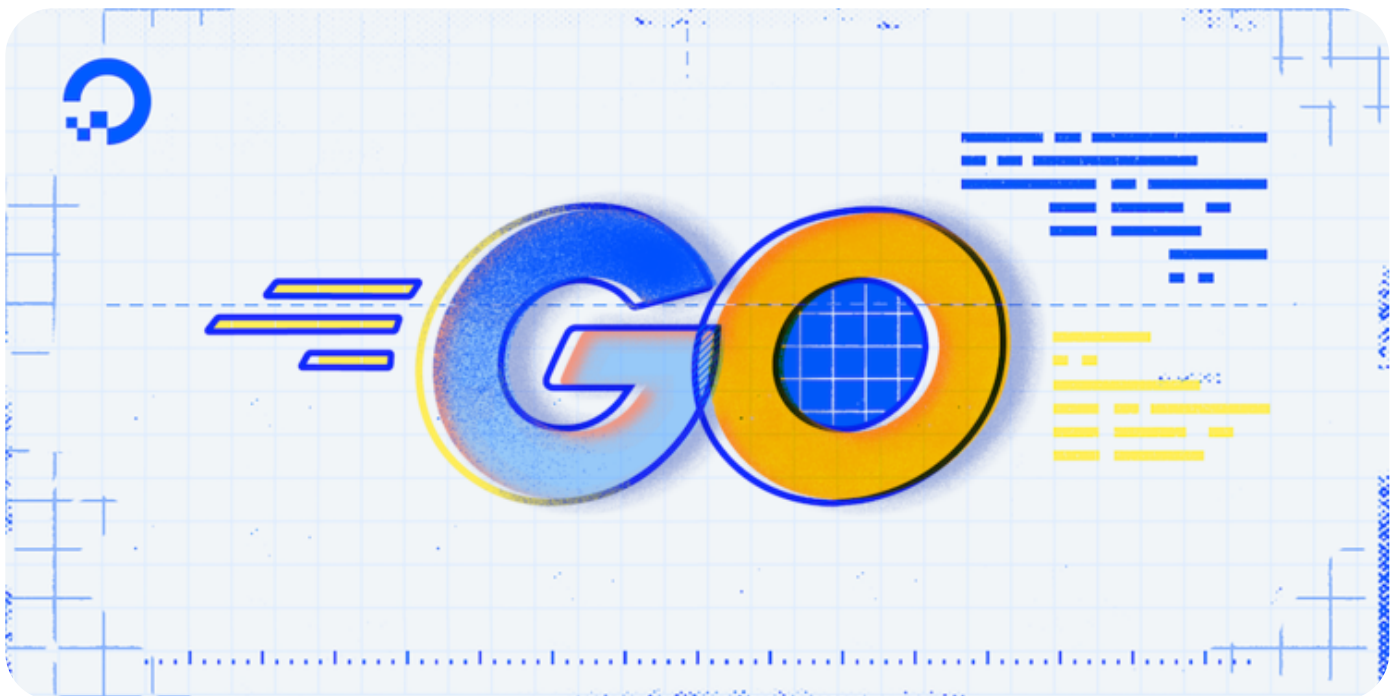
# How To Make an HTTP Server in Go

Published on April 21, 2022

Development   Go



By [Kristin Davidson](#)  
Bit Transducer



The author selected the [Diversity in Tech Fund](#) to receive a donation as part of the [Write](#)

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

it's a request for a cat image or a request to load the tutorial you're reading now. The Go standard library provides built-in support for creating an HTTP server to serve your web content or making HTTP requests to those servers.

In this tutorial, you will create an HTTP server using Go's standard library and then expand your server to read data from the request's query string, the body, and form data. You'll also update your program to respond to the request with your own HTTP headers and status codes.

## # Prerequisites

To follow this tutorial, you will need:

- Go version 1.16 or greater installed. To set this up, follow the [How To Install Go](#) tutorial for your operating system.
- Ability to use [curl](#) to make web requests. To read up on curl, check out [How To Download Files with cURL](#).
- Familiarity with using JSON in Go, which can be found in the [How To Use JSON in Go](#) tutorial.
- Experience with Go's `context` package, which can be attained in the tutorial, [How To Use Contexts in Go](#).
- Experience running goroutines and reading channels, which can be gained from the tutorial, [How To Run Multiple Functions Concurrently in Go](#).
- Familiarity with how [HTTP](#) requests are composed and sent (recommended).

## # Setting Up the Project

In Go, most of the HTTP functionality is provided by the `net/http` package in the standard library, while the rest of the network communication is provided by the `net` package. The `net/http` package not only includes the ability to make HTTP requests, but also provides an HTTP server you can use to handle those requests.

In this section, you will create a program that uses the `http.ListenAndServe` function to start an HTTP server that responds to the request paths `/` and `/hello`. Then, you will expand that program to run multiple HTTP servers in the same program.

Before you write any code, though, you'll need to get your program's directory created

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```
$ mkdir projects
$ cd projects
```

[Copy](#)

Next, make the directory for your project and navigate to that directory. In this case, use the directory `httpserver`:

```
$ mkdir httpserver
$ cd httpserver
```

[Copy](#)

Now that you have your program's directory created and you're in the `httpserver` directory, you can start implementing your HTTP server.

## # Listening for Requests and Serving Responses

A Go HTTP server includes two major components: the server that listens for requests coming from HTTP clients and one or more request handlers that will respond to those requests. In this section, you'll start by using the function `http.HandleFunc` to tell the server which function to call to handle a request to the server. Then, you'll use the `http.ListenAndServe` function to start the server and tell it to listen for new HTTP requests and then serve them using the handler functions you set up.

Now, inside the `httpserver` directory you created, use `nano`, or your favorite editor, to open the `main.go` file:

```
$ nano main.go
```

[Copy](#)

In the `main.go` file, you will create two functions, `getRoot` and `getHello`, to act as your handler functions. Then, you'll create a `main` function and use it to set up your request handlers with the `http.HandleFunc` function by passing it the `/` path for the `getRoot` handler function and the `/hello` path for the `getHello` handler function. Once you've set up your handlers, call the `http.ListenAndServe` function to start the server and listen for requests.

Add the following code to the file to start your program and set up the handlers:

main.go

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```
"io"
"net/http"
"os"

)

func getRoot(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("got / request\n")
    io.WriteString(w, "This is my website!\n")
}
func getHello(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("got /hello request\n")
    io.WriteString(w, "Hello, HTTP!\n")
}
```

In this first chunk of code, you set up the `package` for your Go program, `import` the required packages for your program, and create two functions: the `getRoot` function and the `getHello` function. Both of these functions have the same function signature, where they accept the same arguments: an `http.ResponseWriter` value and an `*http.Request` value. This function signature is used for HTTP handler functions and is defined as [http.HandlerFunc](#). When a request is made to the server, it sets up these two values with information about the request being made and then calls the handler function with those values.

In an `http.HandlerFunc`, the [http.ResponseWriter](#) value (named `w` in your handlers) is used to control the response information being written back to the client that made the request, such as the body of the response or the status code. Then, the [\\*http.Request](#) value (named `r` in your handlers) is used to get information about the request that came into the server, such as the body being sent in the case of a `POST` request or information about the client that made the request.

For now, in both of your HTTP handlers, you use `fmt.Printf` to print when a request comes in for the handler function, then you use the `http.ResponseWriter` to send some text to the response body. The `http.ResponseWriter` is an [io.Writer](#), which means you can use anything capable of writing to that interface to write to the response body. In this case, you're using the [io.WriteString](#) function to write your response to the body.

Now, continue creating your program by starting your `main` function:

main.go

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```
err := http.ListenAndServe(":3333", nil)
```

```
...
```

In the `main` function, you have two calls to the `http.HandleFunc` function. Each call to the function sets up a handler function for a specific request path in the default server multiplexer. The server multiplexer is an `http.Handler` that is able to look at a request path and call a given handler function associated with that path. So, in your program, you're telling the default server multiplexer to call the `getRoot` function when someone requests the `/` path and the `getHello` function when someone requests the `/hello` path.

Once the handlers are set up, you call the `http.ListenAndServe` function, which tells the global HTTP server to listen for incoming requests on a specific port with an optional `http.Handler`. In your program, you tell the server to listen on `":3333"`. By not specifying an IP address before the colon, the server will listen on every IP address associated with your computer, and it will listen on port `3333`. A [network port](#), such as `3333` here, is a way for one computer to have many programs communicating with each other at the same time. Each program uses its own port, so when a client connects to a specific port the computer knows which program to send it to. If you wanted to only allow connections to `localhost`, the hostname for IP address `127.0.0.1`, you could instead say `127.0.0.1:3333`.

Your `http.ListenAndServe` function also passes a `nil` value for the `http.Handler` parameter. This tells the `ListenAndServe` function that you want to use the default server multiplexer and not the one you've set up.

The `ListenAndServe` is a blocking call, which means your program won't continue running until after `ListenAndServe` finishes running. However, `ListenAndServe` won't finish running until your program finishes running or the HTTP server is told to shut down. Even though `ListenAndServe` is blocking and your program doesn't include a way to shut down the server, it's still important to include error handling because there are a few ways calling `ListenAndServe` can fail. So, add error handling to your `ListenAndServe` in the `main` function as shown:

main.go

```
...
```

Copy

```
func main() {
```

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```
        os.Exit(1)
    }
}
```

The first error you're checking for, `http.ErrServerClosed`, is returned when the server is told to shut down or close. This is usually an expected error because you'll be shutting down the server yourself, but it can also be used to show why the server stopped in the output. In the second error check, you check for any other error. If this happens, it will print the error to the screen and then exit the program with an error code of 1 using the `os.Exit` function.

One error you may see while running your program is the `address already in use` error. This error can be returned when `ListenAndServe` is unable to listen on the address or port you've provided because another program is already using it. Sometimes this can happen if the port is commonly used and another program on your computer is using it, but it can also happen if you run multiple copies of your own program multiple times. If you see this error as you're working on this tutorial, make sure you've stopped your program from a previous step before running it again.

**Note:** If you see the `address already in use` error and you don't have another copy of your program running, it could mean some other program is using it. If this happens, wherever you see 3333 mentioned in this tutorial, change it to another number above 1024 and below 65535, such as 3334, and try again. If you still see the error you may need to keep trying to find a port that isn't being used. Once you find a port that works, use that going forward for all your commands in this tutorial.

Now that your code is ready, save your `main.go` file and run your program using `go run`. Unlike other Go programs you may have written, this program won't exit right away on its own. Once you run the program, continue to the next commands:

```
$ go run main.go
```

[Copy](#)

Since your program is still running in your terminal, you will need to open a second terminal to interact with your server. When you see commands or output with the same color as the command below, it means to run it in this second terminal.

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.



requests. Your server is listening for connections on your computer's port 3333 , so you'll want to make your request to `localhost` on that same port:

```
$ curl http://localhost:3333
```

[Copy](#)

The output will look like this:

Output

```
This is my website!
```

In the output you'll see the `This is my website!` response from the `getRoot` function, because you accessed the `/` path on your HTTP server.

Now, in that same terminal, make a request to the same host and port, but add the `/hello` path to the end of your `curl` command:

```
$ curl http://localhost:3333/hello
```

[Copy](#)

Your output will look like this:

Output

```
Hello, HTTP!
```

This time you'll see the `Hello, HTTP!` response from the `getHello` function.

If you refer back to the terminal you have your HTTP server function running in, you now have two lines of output from your server. One for the `/` request and another for the `/hello` request:

Output

```
got / request  
got /hello request
```

Since the server will continue running until the program finishes running, you'll need to stop it yourself. To do this. press `CONTROL+C` to send your program the interrupt signal to stop it.

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.



the bug because it might only exist if certain functions are called in a particular order. So, to avoid this problem, you'll update your server to use a server multiplexer you create yourself in the next section.

## # Multiplexing Request Handlers

When you started your HTTP server in the last section, you passed the `ListenAndServe` function a `nil` value for the `http.Handler` parameter because you were using the default server multiplexer. Because `http.Handler` is an [interface](#), it's possible to create your own struct that implements the interface. But sometimes, you only need a basic `http.Handler` that calls a single function for a specific request path, like the default server multiplexer. In this section, you will update your program to use [http.ServeMux](#), a server multiplexer and `http.Handler` implementation provided by the `net/http` package, which you can use for these cases.

The `http.ServeMux` struct can be configured the same as the default server multiplexer, so there aren't many updates you need to make to your program to start using your own instead of a global default. To update your program to use an `http.ServeMux`, open your `main.go` file again and update your program to use your own `http.ServeMux`:

main.go

Copy

```
...

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", getRoot)
    mux.HandleFunc("/hello", getHello)

    err := http.ListenAndServe(":3333", mux)

    ...
}
```

In this update, you created a new `http.ServeMux` using the `http.NewServeMux` constructor and assigned it to the `mux` variable. After that, you only needed to update the `http.HandleFunc` calls to use the `mux` variable instead of calling the `http` package. Finally,

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```
$ go run main.go
```

[Copy](#)

Your program will continue running as it did last time, so you'll need to run commands to interact with the server in another terminal. First, use `curl` to request the `/` path again:

```
$ curl http://localhost:3333
```

[Copy](#)

The output will look like the following:

Output

```
This is my website!
```

You'll see this output is the same as before.

Next, run the same command from before for the `/hello` path:

```
$ curl http://localhost:3333/hello
```

[Copy](#)

The output will look like this:

Output

```
Hello, HTTP!
```

The output for this path is the same as before as well.

Finally, if you refer back to the original terminal you'll see the output for both the `/` and `/hello` requests as before:

Output

```
got / request  
got /hello request
```

The update you made to the program is functionally the same, but this time you're using your own `http.Handler` instead of the default one.

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

In addition to using your own `http.Handler`, the Go `net/http` package also allows you to use an HTTP server other than the default one. Sometimes you may want to customize how the server runs, or you may want to run multiple HTTP servers in the same program at once. For example, you may have a public website and a private admin website you want to run from the same program. Since you can only have one default HTTP server, you wouldn't be able to do this with the default one. In this section, you will update your program to use two `http.Server` values provided by the `net/http` package for cases like these — when you want more control over the server or need multiple servers at the same time.

In your `main.go` file, you will set up multiple HTTP servers using `http.Server`. You'll also update your handler functions to access the `context.Context` for the incoming `*http.Request`. This will allow you to set which server the request is coming from in a `context.Context` variable, so you can print the server in the handler function's output.

Open your `main.go` file again and update it as shown:

main.go

Copy

```
package main

import (
    // Note: Also remove the 'os' import.
    "context"
    "errors"
    "fmt"
    "io"
    "net"
    "net/http"
)

const keyServerAddr = "serverAddr"

func getRoot(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    fmt.Printf("%s: got / request\n", ctx.Value(keyServerAddr))
    io.WriteString(w, "This is my website!\n")
}

func getHello(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
```

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

In the code update above, you updated the `import` statement to include the packages needed for the update. Then, you created a `const string` value called `keyServerAddr` to act as the key for the HTTP server's address value in the `http.Request` context. Lastly, you updated both your `getRoot` and `getHello` functions to access the `http.Request`'s `context.Context` value. Once you have the value, you include the HTTP server's address in the `fmt.Printf` output so you can see which of the two servers handled the HTTP request.

Now, start updating your `main` function by adding the first of your two `http.Server` values:

main.go

Copy

```
...
func main() {
    ...
    mux.HandleFunc("/hello", getHello)

    ctx, cancelCtx := context.WithCancel(context.Background())
    serverOne := &http.Server{
        Addr:    ":3333",
        Handler: mux,
        BaseContext: func(l net.Listener) context.Context {
            ctx = context.WithValue(ctx, keyServerAddr, l.Addr().String())
            return ctx
        },
    }
}
```

In the updated code, the first thing you did is create a new `context.Context` value with an available function, `cancelCtx`, to cancel the context. Then, you define your `serverOne` `http.Server` value. This value is very similar to the HTTP server you've already been using, but instead of passing the address and the handler to the `http.ListenAndServe` function, you set them as the `Addr` and `Handler` values of the `http.Server`.

One other change is adding a `BaseContext` function. `BaseContext` is a way to change parts of the `context.Context` that handler functions receive when they call the `Context` method of `*http.Request`. In the case of your program, you're adding the address the server is listening on (`l.Addr().String()`) to the context with the key `serverAddr`, which will then be printed to the handler function's output.

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```
func main() {
    ...
    serverOne := &http.Server {
        ...
    }

    serverTwo := &http.Server{
        Addr:    ":4444",
        Handler: mux,
       BaseContext: func(l net.Listener) context.Context {
            ctx = context.WithValue(ctx, keyServerAddr, l.Addr().String())
            return ctx
        },
    }
}
```

This server is defined the same way as the first server, except instead of `:3333` for the `Addr` field you set it to `:4444`. This way one server will be listening for connections on port `3333` and the second server will listen on port `4444`.

Now, update your program to start the first server, `serverOne`, in a goroutine:

main.go

Copy

```
...

func main() {
    ...
    serverTwo := &http.Server {
        ...
    }

    go func() {
        err := serverOne.ListenAndServe()
        if errors.Is(err, http.ErrServerClosed) {
            fmt.Printf("server one closed\n")
        } else if err != nil {
            fmt.Printf("error listening for server one: %s\n", err)
        }
        cancelCtx()
    }()
}
```

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

cancel the context being provided to the HTTP handlers and both server `BaseContext` functions. This way, if the server ends for some reason, the context will end as well.

Finally, update your program to start the second server in a goroutine as well:

main.go

Copy

```
...

func main() {
    ...
    go func() {
        ...
    }()
    go func() {
        err := serverTwo.ListenAndServe()
        if errors.Is(err, http.ErrServerClosed) {
            fmt.Printf("server two closed\n")
        } else if err != nil {
            fmt.Printf("error listening for server two: %s\n", err)
        }
        cancelCtx()
    }()

    <-ctx.Done()
}
```

This goroutine is functionally the same as the first one, it just starts `serverTwo` instead of `serverOne`. This update also includes the end of the `main` function where you read from the `ctx.Done` channel before returning from the `main` function. This ensures that your program will stay running until either of the server goroutines ends and `cancelCtx` is called. Once the context is over, your program will exit.

Save and close the file when you're done.

Run your server using the `go run` command:

```
$ go run main.go
```

Copy

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```
$ curl http://localhost:3333/hello
```

The output will look like this:

Output

```
This is my website!  
Hello, HTTP!
```

In the output you'll see the same responses you saw before.

Now, run those same commands again, but this time use port 4444 , the one that corresponds to `serverTwo` in your program:

```
$ curl http://localhost:4444  
$ curl http://localhost:4444/hello
```

Copy

The output will look like the following:

Output

```
This is my website!  
Hello, HTTP!
```

You'll see the same output for these requests as you did for the requests on port 3333 being served by `serverOne` .

Finally, look back at the original terminal where your server is running:

Output

```
[::]:3333: got / request  
[::]:3333: got /hello request  
[::]:4444: got / request  
[::]:4444: got /hello request
```

The output looks similar to what you saw before, but this time it shows the server that responded to the request. The first two requests show they came from the server listening on port 3333 (`serverOne`), and the second two requests came from the server listening on

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.



not, you'll see `0.0.0.0` instead of `[::]`. The reason for this is that your computer will communicate with itself over IPv6 if configured, and `[::]` is the IPv6 notation for `0.0.0.0`.

Once you're done, use `CONTROL+C` again to stop the server.

In this section, you created a new HTTP server program using `http.HandleFunc` and `http.ListenAndServe` to run and configure the default server. Then, you updated it to use an `http.ServeMux` for the `http.Handler` instead of the default server multiplexer. Finally, you updated your program to use `http.Server` to run multiple HTTP servers in the same program.

While you have an HTTP server running now, it's not very interactive. You can add new paths that it responds to, but there's not really a way for users to interact with it past that. The HTTP protocol includes a number of ways users can interact with an HTTP server beyond paths. In the next section, you'll update your program to support the first of them: query string values.

## # Inspecting a Request's Query String

One of the ways a user is able to influence the HTTP response they get back from an HTTP server is by using the [query string](#). The query string is a set of values added to the end of a [URL](#). It starts with a `?` character, with additional values added using `&` as a delimiter. Query string values are commonly used as a way to filter or customize the results an HTTP server sends as a response. For example, one server may use a `results` value to allow a user to specify something like `results=10` to say they'd like to see 10 items in their list of results.

In this section, you will update your `getRoot` handler function to use its `*http.Request` value to access query string values and print them to the output.

First, open your `main.go` file and update the `getRoot` function to access the query string with the `r.URL.Query` method. Then, update the `main` method to remove `serverTwo` and all its associated code since you'll no longer need it:

main.go

...

Copy

```
func getRoot(w http.ResponseWriter, r *http.Request) {
```

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```

second := r.URL.Query().Get("second")

fmt.Printf("%s: got / request. first(%t)=%s, second(%t)=%s\n",
    ctx.Value(keyServerAddr),
    hasFirst, first,
    hasSecond, second)
io.WriteString(w, "This is my website!\n")
}
...

```

In the `getRoot` function, you use the `r.URL` field of `getRoot`'s `*http.Request` to access properties about the URL being requested. Then you use the `Query` method of the `r.URL` field to access the query string values of the request. Once you're accessing the query string values, there are two methods you can use to interact with the data. The `Has` method returns a `bool` value specifying whether the query string has a value with the key provided, such as `first`. Then, the `Get` method returns a `string` with the value of the key provided.

In theory, you could always use the `Get` method to retrieve query string values because it will always return either the actual value for the given key or an empty string if the key doesn't exist. For many uses, this is good enough — but in some cases, you may want to know the difference between a user providing an empty value or not providing a value at all. Depending on your use case, you may want to know whether a user has provided a `filter` value of nothing, or if they didn't provide a `filter` at all. If they provided a `filter` value of nothing you may want to treat it as "don't show me anything," whereas not providing a `filter` value would mean "show me everything." Using `Has` and `Get` will allow you to tell the difference between these two cases.

In your `getRoot` function, you also updated the output to show the `Has` and `Get` values for both `first` and `second` query string values.

Now, update your `main` function to go back to using one server again:

main.go

...

Copy

```
func main() {
```

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```

BaseContext: func(l net.Listener) context.Context {
    ctx = context.WithValue(ctx, keyServerAddr, l.Addr().String())
    return ctx
},
}

err := server.ListenAndServe()
if errors.Is(err, http.ErrServerClosed) {
    fmt.Printf("server closed\n")
} else if err != nil {
    fmt.Printf("error listening for server: %s\n", err)
}
}

```

In the `main` function, you removed references to `serverTwo` and moved running the `server` (formerly `serverOne`) out of a goroutine and into the `main` function, similar to how you were running `http.ListenAndServe` earlier. You could also change it back to `http.ListenAndServe` instead of using an `http.Server` value since you only have one server running again, but by using `http.Server`, you would have less to update if you wanted to make any additional customizations to the server in the future.

Now, once you've saved your changes, use `go run` to run your program again:

```
$ go run main.go
```

Copy

Your server will start running again, so swap back to your second terminal to run a `curl` command with a query string. In this command you'll need to surround your URL with single quotes ( `'` ), otherwise your terminal's shell may interpret the `&` symbol in the query string as the "run this command in the background" feature that many shells include. In the URL, include a value of `first=1` for `first`, and `second=` for `second`:

```
$ curl 'http://localhost:3333?first=1&second='
```

Copy

The output will look like this:

Output

```
This is my website!
```

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

### Output

```
[::]:3333: got / request. first(true)=1, second(true)=
```

The output for the `first` query string value shows the `Has` method returned `true` because `first` has a value, and also that `Get` returned the value of `1`. The output for `second` shows that `Has` returned `true` because `second` was included, but the `Get` method didn't return anything other than an empty string. You can also try making different requests by adding and removing `first` and `second` or setting different values to see how it changes the output from those functions.

Once you're finished, press `CONTROL+C` to stop your server.

In this section you updated your program to use only one `http.Server` again, but you also added support for reading `first` and `second` values from the query string for the `getRoot` handler function.

Using the query string isn't the only way for users to provide input to an HTTP server, though. Another common way to send data to a server is by including data in the request's body. In the next section, you'll update your program to read a request's body from the `*http.Request` data.

## # Reading a Request Body

When creating an HTTP-based API, such as a [REST API](#), a user may need to send more data than can be included in URL length limits, or your page may need to receive data that isn't about how the data should be interpreted, such as filters or result limits. In these cases, it's common to include data in the request's body and to send it with either a `POST` or a `PUT` HTTP request.

In a Go `http.HandlerFunc`, the `*http.Request` value is used to access information about the incoming request, and it also includes a way to access the request's body with the `Body` field. In this section, you will update your `getRoot` handler function to read the request's body.

To update your `getRoot` method, open your `main.go` file and update it to use `ioutil.ReadAll` to read the `r.Body` request field:

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```

    ...
    "io/ioutil"
    ...
)

...

func getRoot(w http.ResponseWriter, r *http.Request) {
    ...
    second := r.URL.Query().Get("second")

    body, err := ioutil.ReadAll(r.Body)
    if err != nil {
        fmt.Printf("could not read body: %s\n", err)
    }

    fmt.Printf("%s: got / request. first(%t)=%s, second(%t)=%s, body:\n%s\n",
        ctx.Value(keyServerAddr),
        hasFirst, first,
        hasSecond, second,
        body)
    io.WriteString(w, "This is my website!\n")
}

...

```

In this update, you use the `ioutil.ReadAll` function to read the `r.Body` property of the `*http.Request` to access the request's body. The `ioutil.ReadAll` function is a utility function that will read data from an [io.Reader](#) until it encounters an error or the end of the data. Since `r.Body` is an `io.Reader`, you're able to use it to read the body. Once you've read the body, you also updated the `fmt.Printf` to print it to the output.

After you've saved your updates, run your server using the `go run` command:

```
$ go run main.go
```

Copy

Since the server will continue running until you stop it, go to your other terminal to make a `POST` request using `curl` with the `-x POST` option and a body using the `-d` option. You can also use the `first` and `second` query string values from before as well:

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

#### Output

```
This is my website!
```

The output from your handler function is the same, but you'll see your server logs have been updated again:

#### Output

```
[::]:3333: got / request. first(true)=1, second(true)=, body:  
This is the body
```

In the server logs, you'll see the query string values from before, but now you'll also see the `This is the body` data the `curl` command sent.

Now, stop the server by pressing `CONTROL+C`.

In this section, you updated your program to read a request's body into a variable you printed to the output. By combining reading the body this way with other functionality, such as [encoding/json](#) to unmarshal a JSON body into Go data, you'll be able to create APIs your users can interact with in a way they're familiar with from other APIs.

Not all data sent from a user is in the form of an API, though. Many websites have forms they ask their users to fill out, so in the next section, you'll update your program to read form data in addition to the request body and query string you already have.

## # Retrieving Form Data

For a long time, sending data using forms was the standard way for users to send data to an HTTP server and interact with a website. Forms aren't as popular now as they were in the past, but they still have many uses as a way for users to submit data to a website. The `*http.Request` value in `http.HandlerFunc` also provides a way to access this data, similar to how it provides access to the query string and request body. In this section, you'll update your `getHello` program to receive a user's name from a form and respond back to them with their name.

Open your `main.go` and update the `getHello` function to use the `PostFormValue` method of `*http.Request`:

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

```
func getHello(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    fmt.Printf("%s: got /hello request\n", ctx.Value(keyServerAddr))

    myName := r.PostFormValue("myName")
    if myName == "" {
        myName = "HTTP"
    }
    io.WriteString(w, fmt.Sprintf("Hello, %s!\n", myName))
}

...
```

Now in your `getHello` function, you're reading the form values posted to your handler function and looking for a value named `myName`. If the value isn't found or the value found is an empty string, you set the `myName` variable to a default value of `HTTP` so the page doesn't display an empty name. Then, you updated the output to the user to display the name they sent, or `HTTP` if they didn't send a name.

To run your server with these updates, save your changes and run it using `go run`:

```
$ go run main.go
```

Copy

Now, in your second terminal, use `curl` with the `-x POST` option to the `/hello` URL, but this time instead of using `-d` to provide a data body, use the `-F 'myName=Sammy'` option to provide form data with a `myName` field with the value `Sammy`:

```
$ curl -X POST -F 'myName=Sammy' 'http://localhost:3333/hello'
```

Copy

The output will look like this:

Output

```
Hello, Sammy!
```

In the output above, you'll see the expected `Hello, Sammy!` greeting because the form you sent with `curl` said `myName` is `Sammy`.

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.



also remove the `-F` option and include `myName=Sammy` in the query string to see `Sammy` returned as well. If you did that without the change to `r.FormValue`, though, you'd see the default HTTP response for the name. Being careful about where you're retrieving these values from can avoid potential conflicts in names or bugs that are hard to track down. It's useful to be more strict and use the `r.PostFormValue` unless you want the flexibility to put it in the query string as well.

If you look back at your server logs you'll see the `/hello` request was logged similar to previous requests:

#### Output

```
[::]:3333: got /hello request
```

To stop the server, press `CONTROL+C`.

New! Premium CPU-Optimized Droplets are now available. [Learn more](#) →

[We're hiring](#)

[Blog](#)

[Docs](#)

[Get Support](#)

[Sales](#)



[Tutorials](#) [Questions](#) [Learning Paths](#) [For Businesses](#) [For Developers](#) [Social Impact](#)

The HTTP protocol uses a few features that users don't normally see to send data to help browsers or servers communicate. One of these features is called the [status code](#), and is used by the server to give an HTTP client a better idea of whether the server considers the request successful, or whether something went wrong on either the server side, or with the request the client sent.

Another way HTTP servers and clients communicate is by using [header fields](#). A header field is a key and value that either a client or server will send to the other to let them know about themselves. There are many headers that are pre-defined by the HTTP protocol, such as `Accept`, which a client uses to tell the server the type of data it can accept and understand. It's also possible to define your own by prefixing them with `x-` and then the rest of a name.

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

To add this feature to your program, open your `main.go` file one last time and add the validation check to the `getHello` handler function:

`main.go`

Copy

```
...

func getHello(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    fmt.Printf("%s: got /hello request\n", ctx.Value(keyServerAddr))

    myName := r.PostFormValue("myName")
    if myName == "" {
        w.Header().Set("x-missing-field", "myName")
        w.WriteHeader(http.StatusBadRequest)
        return
    }
    io.WriteString(w, fmt.Sprintf("Hello, %s!\n", myName))
}

...
```

In this update, when `myName` is an empty string, instead of setting a default name of HTTP, you send the client an error message instead. First, you use the `w.Header().Set` method to set an `x-missing-field` header with a value of `myName` in the response HTTP headers. Then, you use the `w.WriteHeader` method to write any response headers, as well as a “Bad Request” status code, to the client. Finally, it will return out of the handler function. You want to make sure you do this so you don’t accidentally write a `Hello, !` response to the client in addition to the error information.

It’s also important to be sure you’re setting headers and sending the status code in the right order. In an HTTP request or response, all the headers must be sent before the body is sent to the client, so any requests to update `w.Header()` must be done before `w.WriteHeader` is called. Once `w.WriteHeader` is called, the status of the page is sent with all the headers and only the body can be written to after.

Once you’ve saved your updates, you can run your program again with the `go run` command:

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

`curl` to show verbose output so you can see all the headers and output for the request:

```
$ curl -v -X POST 'http://localhost:3333/hello'
```

[Copy](#)

This time in the output, you'll see a lot more information as the request is processed because of the verbose output:

#### Output

```
* Trying ::1:3333...
* Connected to localhost (::1) port 3333 (#0)
> POST /hello HTTP/1.1
> Host: localhost:3333
> User-Agent: curl/7.77.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 400 Bad Request
< X-Missing-Field: myName
< Date: Wed, 02 Mar 2022 03:51:54 GMT
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

The first couple of lines in the output show that `curl` is trying to connect to `localhost` port `3333`.

Then, the lines that start with `>` show the request `curl` is making to the server. It says `curl` is making a `POST` request to the `/hello` URL using the HTTP 1.1 protocol, as well as a few other headers. Then, it sends an empty body as shown by the empty `>` line.

Once `curl` sends the request, you can see the response it receives from the server with the `<` prefix. The first line says that the server responded with a `Bad Request`, which is also known as a 400 status code. Then, you can see the `X-Missing-Field` header you set is included with a value of `myName`. After sending a few additional headers, the request finishes without sending any of the body, which can be seen by the `Content-Length` (or body) being length `0`.

If you look back at your server output once more, you'll see the `/hello` request the server

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

One last time, press `CONTROL+C` to stop your server.

In this section, you updated your HTTP server to add validation to the `/hello` form input. If a name isn't sent as part of the form, you used `w.Header().Set` to set a header to send back to the client. Once the header is set, you used `w.WriteHeader` to write the headers to the client, as well as a status code indicating to the client it was a bad request.

## # Conclusion

In this tutorial, you created a new Go HTTP server using the `net/http` package in Go's standard library. You then updated your program to use a specific server multiplexer and multiple `http.Server` instances. You also updated your server to read user input via query string values, the request body, and form data. Finally, you updated your server to return form validation information to the client using a custom HTTP header and a "Bad Request" status code.

One good thing about the Go HTTP ecosystem is that many frameworks are designed to integrate neatly into Go's `net/http` package instead of reinventing a lot of the code that already exists. The [github.com/go-chi/chi](https://github.com/go-chi/chi) project is a good example of this. The server multiplexer built into Go is a good way to get started with an HTTP server, but it lacks a lot of advanced functionality a larger web server may need. Projects such as `chi` are able to implement the `http.Handler` interface in the Go standard library to fit right into the standard `http.Server` without needing to rewrite the server portion of the code. This allows them to focus on creating [middleware](#) and other tooling to enhance what's available instead of working on the basic functionality.

In addition to projects like `chi`, the Go `net/http` package also includes a lot of functionality not covered in this tutorial. To explore more about working with cookies or serving HTTPS traffic, the `net/http` package is a good place to start.

This tutorial is also part of the [DigitalOcean How to Code in Go](#) series. The series covers a number of Go topics, from installing Go for the first time to how to use the language itself.

Thanks for learning with the DigitalOcean Community. Check out our offerings for compute, storage, networking, and managed databases.

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

[Next in series: How To Make HTTP Requests in Go →](#)

## Get \$200 to try DigitalOcean – and do all the below for free!

Build applications, host websites, run open source software, learn cloud computing, and more – every cloud resource you need. If you've never tried DigitalOcean's products or services before, we'll cover your first \$200 in the next 60 days.

[Sign up now to activate this offer →](#)

## Tutorial Series: How To Code in Go

Go (or GoLang) is a modern programming language originally developed by Google that uses high-level syntax similar to scripting languages. It is popular for its minimal syntax and innovative handling of concurrency, as well as for the tools it provides for building native binaries on foreign platforms.

Subscribe

Development   Go

## Browse Series: 53 articles

[1/53 How To Code in Go eBook](#)

[2/53 How To Install Go and Set Up a Local Programming Environment on Ubuntu 18.04](#)

[3/53 How To Install Go and Set Up a Local Programming Environment on macOS](#)

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

## About the authors



[Kristin Davidson](#) Author

Bit Transducer

Kristin is a life-long geek and enjoys digging into the lowest levels of computing. She also enjoys learning and tinkering with new technologies.



[Rachel Lee](#) Editor

Technical Editor

Still looking for an answer?

Ask a question

Search for more help

Was this helpful?

Yes

No



## Comments

### 1 Comments

Rich text editor toolbar with icons for bold, italic, underline, link, unlink, list, quote, code, and other formatting options.

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

This textbox defaults to using **Markdown** to format your answer.

You can type **!ref** in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

[Sign In](#) or [Sign Up](#) to Comment

[Justin Merrell](#) • October 28, 2022



Is this extra? "<^>"

[Reply](#)



This work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License.

**Try DigitalOcean for free**

Click below to sign up and get **\$200 of credit** to try our products over 60 days!

[Sign up](#) →




## Popular Topics

... .

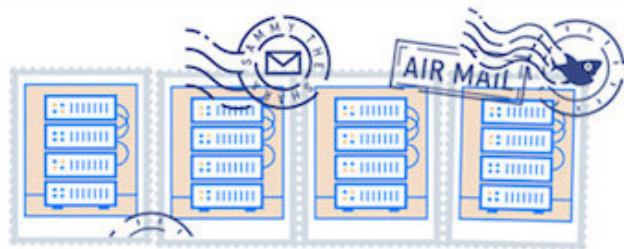
This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.



[MySQL](#)[Docker](#)[Kubernetes](#)[All tutorials →](#)[Free Managed Hosting →](#)

-  Congratulations on unlocking the whale ambience easter egg! Click the whale button in the bottom left of your screen to toggle some ambient whale noises while you read.
-  Thank you to the [Glacier Bay National Park & Preserve](#) and [Merrick079](#) for the sounds behind this easter egg.
-  Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the [Whale and Dolphin Conservation](#).

[Reset easter egg to be discovered again](#) / [Permanently dismiss and hide easter egg](#)



## Get our biweekly newsletter

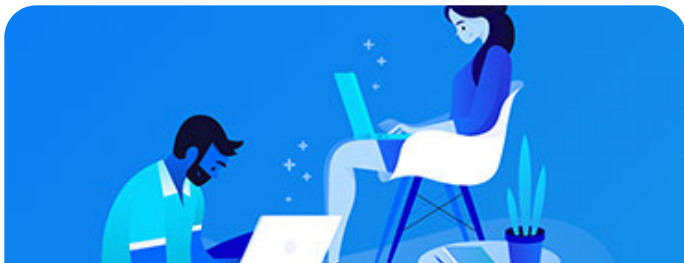
Sign up for Infrastructure as a Newsletter.



## Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

This site uses cookies and related technologies, as described in our [privacy policy](#), for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.



## Become a contributor

You get paid; we donate to tech nonprofits.

[Learn more](#) →

## Featured on Community

[Kubernetes Course](#)

[Learn Python 3](#)

[Machine Learning in Python](#)

[Getting started with Go](#)

[Intro to Kubernetes](#)

## DigitalOcean Products

[Cloudways](#)

[Virtual Machines](#)

[Managed Databases](#)

[Managed Kubernetes](#)

[Block Storage](#)

[Object Storage](#)

[Marketplace](#)

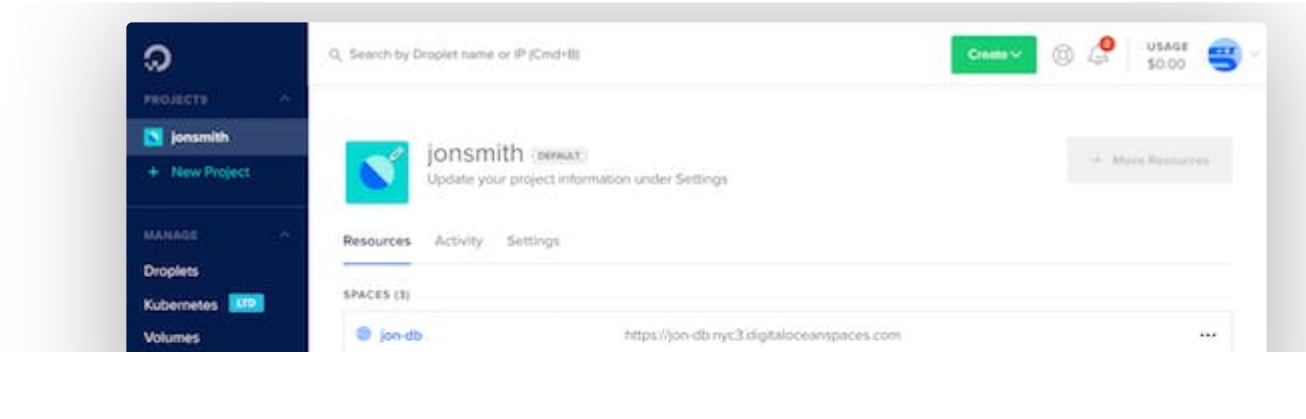
[VPC](#)

[Load Balancers](#)

## Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale

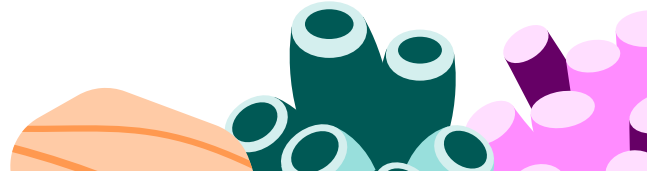
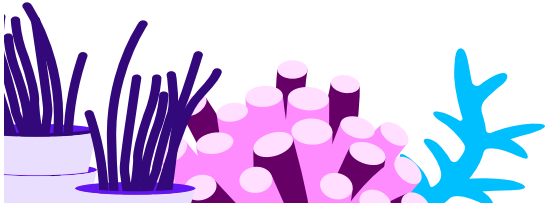
This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.



Company	Products	Community	Solutions	Contact
About	Products Overview	Tutorials	Website Hosting	Support
Leadership		Q&A	VPS Hosting	Sales
Blog	Droplets	CSS-Tricks	Web & Mobile Apps	Report Abuse
Careers	Kubernetes	Write for DOnations	Game Development	System Status
Customers	App Platform	Currents Research	Streaming	Share your ideas
Partners	Functions	Hatch Startup Program	VPN	
Channel Partners	Managed Databases	deploy by DigitalOcean	SaaS Platforms	
Referral Program	Spaces	Shop Swag	Cloud Hosting for Blockchain	
Affiliate Program	Marketplace	Research Program	Startup Resources	
Press	Load Balancers	Open Source		
Legal	Block Storage	Code of Conduct		
Security	Tools & Integrations	Newsletter Signup		
Investor Relations	API	Meetups		
DO Impact	Pricing			
	Documentation			
	Release Notes			

This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.

© 2023 DigitalOcean, LLC. All rights reserved.



This site uses cookies and related technologies, as described in our privacy policy, for purposes that may include site operation, analytics, enhanced user experience, or advertising. You may choose to consent to our use of these technologies, or manage your own preferences.