MANNING

SAMPLE CHAPTER

Go Web Programming

Sau Sheong Chang

Go Web Programming

by Sau Sheong Chang

Sample Chapter 2

Copyright 2016 Manning Publications

brief contents

PART 1 GO AND WEB APPLICATIONS 1

1 ▪ Go and web applications 3
2 ▪ Go ChitChat 22

PART 2 BASIC WEB APPLICATIONS 45

3 ▪ Handling requests 47
4 ▪ Processing requests 69
5 ▪ Displaying content 96
6 ▪ Storing data 125

PART 3 BEING REAL 153

7 ▪ Go web services 155
8 ▪ Testing your application 190
9 ▪ Leveraging Go concurrency 223
10 ▪ Deploying Go 256

2 Go ChitChat

This chapter covers

- Introducing Go web programming
- Designing a typical Go web application
- Writing a complete Go web application
- Understanding the parts of a Go web application

Toward the end of chapter 1, we went through the simplest possible Go web application. That simple web application, I admit, is pretty useless and is nothing more than the equivalent of a Hello World application. In this chapter, we'll explore another basic but more useful web application. We'll be building a simple internet forum web application—one that allows users to log in and create conversations and respond to conversation topics.

By the end of the chapter, you might not have the skills to write a full-fledged web application but you'll be able to appreciate how one can be structured and developed. Throughout this chapter you'll see the bigger picture of how web applications can be written in Go.

If you find this chapter a bit too intimidating—especially with the rush of Go code—don't be too alarmed. Work through the next few chapters and then revisit this one and you'll find that things become a lot clearer!

2.1 Let's ChitChat

Internet forums are everywhere. They're one of the most popular uses of the internet, related to the older bulletin board systems (BBS), Usenet, and electronic mailing lists. Yahoo! and Google Groups are very popular (see figure 2.1), with Yahoo! reporting 10 million groups (each group is a forum on its own) and 115 million group members. One of the biggest internet forums around, Gaia Online, has 23 million registered users and a million posts made every day, with close to 2 billion posts and counting. Despite the introduction of social networks like Facebook, internet forums remain one of the most widely used means of communications on the internet.

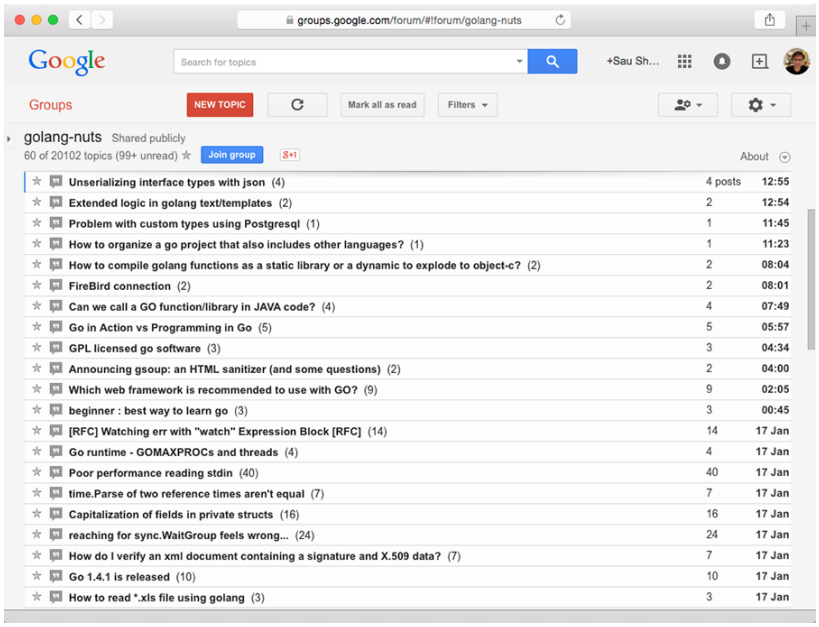

Figure 2.1 Google Groups Go programming language forum, an example of an internet forum

Essentially, internet forums are the equivalent of a giant bulletin board where anyone (either registered or anonymous users) can hold conversations by posting messages on the forum. These conversations, called *threads*, usually start off as a topic that a user wants to talk about, and other users add to the conversation by posting their replies to the original topic. More sophisticated forums are hierarchical, with forums having subforums with specific categories of topics that are being discussed. Most forums are moderated by one or more users, called *moderators*, who have special permissions.

In this chapter, we'll develop a simple internet forum called ChitChat. Because this is a simple example, we'll be implementing only the key features of an internet forum. Users will only be able to sign up for an account and log in to create a thread or post a

reply to an existing thread. A nonregistered user will be able to read the threads but not add new threads or post to existing ones. Let's start off with the application design.

> **Code for this chapter**
>
> Unlike with the other chapters in this book, you won't see all the code that's written for ChitChat here (that would be too much!). But you can check out the entire application on GitHub at https://github.com/sausheong/gwp. If you're planning to run through the exercises while you read this chapter, you'll have an easier time if you get the code from the repository first.

2.2 Application design

ChitChat's application design is typical of any web application. As mentioned in chapter 1, web applications have the general flow of the client sending a request to a server, and a server responding to that request (figure 2.2).

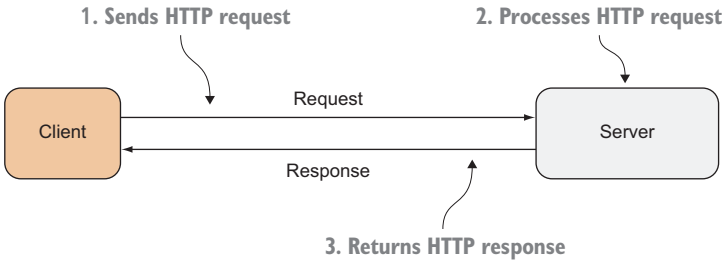

Figure 2.2 How a web application generally works, with a client sending a request to the server and waiting to receive a response

ChitChat's application logic is coded in the server. While the client triggers the requests and provides the data to the server, the format and the data requested are suggested by the server, provided in hyperlinks on the HTML pages that the server serves to the client (figure 2.3).

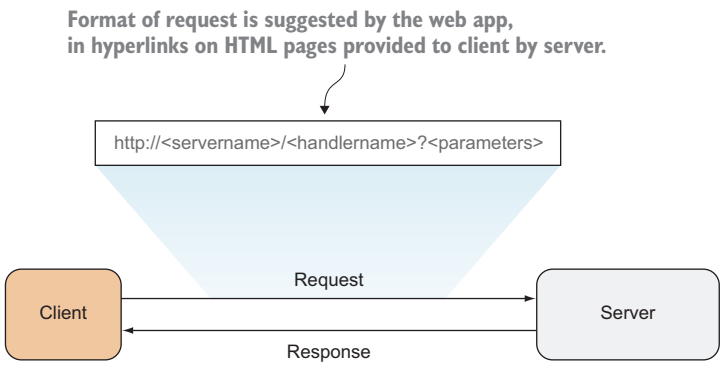

Figure 2.3 The URL format of an HTTP request

The format for the request is normally the prerogative of the application itself. For ChitChat, we'll be using the following format: http://<servername>/<handler-name>?<parameters>

The *server name* is the name of the ChitChat server; the *handler name* is the name of the handler that's being called. The handler name is hierarchical: the root of the handler name is the module that's being called, the second part the submodule, and so on, until it hits the leaf, which is the handler of the request within that submodule. If we have a module called `thread` and we need to have a handler to read the thread, the handler name is `/thread/read`.

The *parameters* of the application, which are URL queries, are whatever we need to pass to the handler to process the request. In this example, we need to provide the unique identifier (ID) of the thread to the handler, so the parameters will be `id=123`, where `123` is the unique ID.

Let's recap the request; this is how the URL being sent into the ChitChat server will look (assuming `chitchat` is the server name): http://chitchat/thread/read?id=123.

When the request reaches the server, a *multiplexer* will inspect the URL being requested and redirect the request to the correct handler. Once the request reaches a handler, the handler will retrieve information from the request and process it accordingly (figure 2.4). When the processing is complete, the handler passes the data to the template engine, which will use templates to generate HTML to be returned to the client.

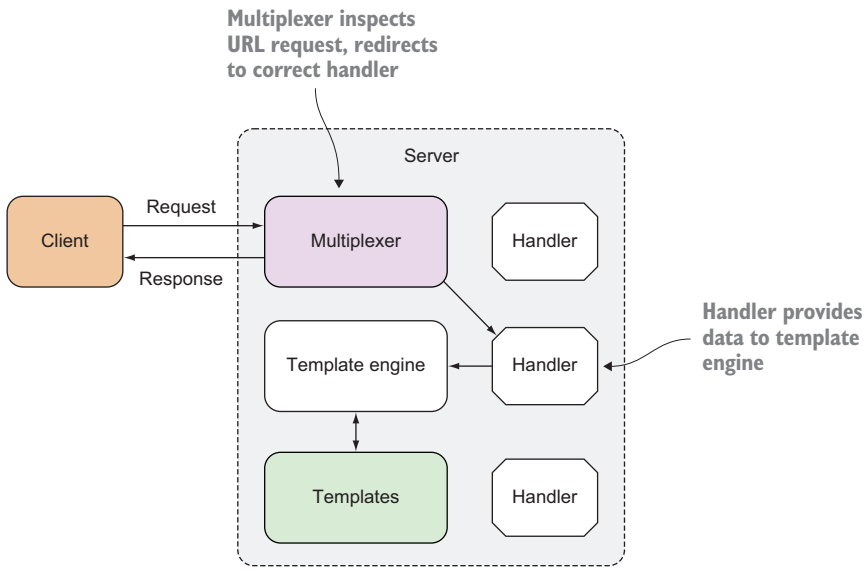

Figure 2.4 How the server works in a typical web application

2.3 Data model

Most applications need to work on data, in one form or another. In ChitChat, we store the data in a relational database (we use PostgreSQL in this book) and use SQL to interact with the database.

ChitChat's data model is simple and consists of only four data structures, which in turn map to a relational database. The four data structures are

- *User*—Representing the forum user's information
- *Session*—Representing a user's current login session
- *Thread*—Representing a forum thread (a conversation among forum users)
- *Post*—Representing a post (a message added by a forum user) within a thread

We'll have users who can log into the system to create and post to threads. Anonymous users can read but won't be able to create threads or posts. To simplify the application, we'll have only one type of user—there are no moderators to approve new threads or posts (figure 2.5).

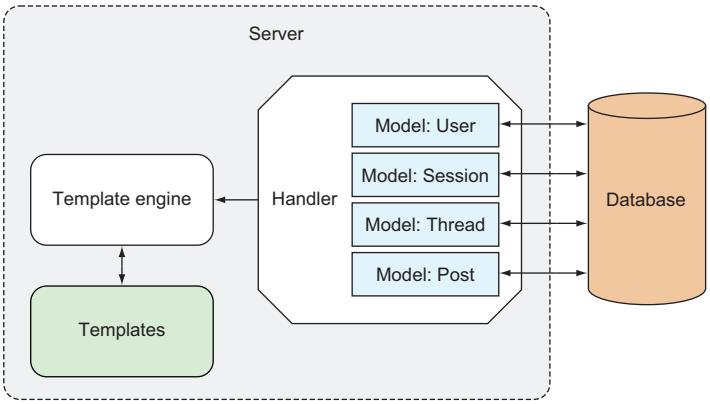

Figure 2.5 How a web application can access the data store

With our application design firmly in mind, let's move on to code. A bit of caution before we begin: there will be code in this chapter that might seem puzzling. If you're a new Go programmer, it might be worth your while to refresh your memory going through an introductory Go programming book like *Go in Action* by William Kennedy with Brian Ketelsen and Erik St. Martin (Manning, 2015).

Otherwise, please hang on; this chapter provides an overall picture of how a Go web application will look but is thin on details. The details will come in the later chapters. Where possible, I'll mention which chapters explore those details as we move along.

2.4 Receiving and processing requests

Receiving and processing requests is the heart of any web application. Let's recap what you've learned so far:

1. A client sends a request to a URL at the server.
2. The server has a multiplexer, which redirects the request to the correct handler to process the request.
3. The handler processes the request and performs the necessary work.
4. The handler calls the template engine to generate the correct HTML to send back to the client.

Let's begin at the beginning, which is the root URL (/). When you type http://localhost, this is where the application will take you. In the next few subsections, we'll discuss how to handle a request to this URL and respond with dynamically generated HTML.

2.4.1 The multiplexer

We start all Go applications with a main source code file, which is the file that contains the `main` function and is the starting point where the compiled binary executes. In ChitChat we call this file main.go.

Listing 2.1 A simple `main` function in main.go

```
package main

import (
  "net/http"
)

func main() {


  mux := http.NewServeMux()
  files := http.FileServer(http.Dir("/public"))
  mux.Handle("/static/", http.StripPrefix("/static/", files))

    mux.HandleFunc("/", index)


  server := &http.Server{
    Addr:     "0.0.0.0:8080",
    Handler:  mux,
  }
  server.ListenAndServe()
}
```

In main.go, you first create a *multiplexer*, the piece of code that redirects a request to a handler. The net/http standard library provides a default multiplexer that can be created by calling the `NewServeMux` function:

```
mux := http.NewServeMux()
```

To redirect the root URL to a handler function, you use the `HandleFunc` function:

```
mux.HandleFunc("/", index)
```

`HandleFunc` takes the URL as the first parameter, and the name of the handler function as the second parameter, so when a request comes for the root URL (`/`), it's redirected to a handler function named `index`. You don't need to provide the parameters to the handler function because all handler functions take `ResponseWriter` as the first parameter and a pointer to `Request` as the second parameter.

Notice that I've done some sleight-of-hand when talking about handlers. I started off talking about handlers and then switched to talking about handler functions. This is intentional; handlers and handler functions are *not* the same, though they provide the same results in the end. We'll talk more about them in chapter 3, but for now let's move on.

2.4.2 Serving static files

Besides redirecting to the appropriate handler, you can use the multiplexer to serve static files. To do this, you use the `FileServer` function to create a handler that will serve files from a given directory. Then you pass the handler to the `Handle` function of the multiplexer. You use the `StripPrefix` function to remove the given prefix from the request URL's path.

```
files := http.FileServer(http.Dir("/public"))
mux.Handle("/static/", http.StripPrefix("/static/", files))
```

In this code, you're telling the server that for all request URLs starting with /static/, strip off the string `/static/` from the URL, and then look for a file with the name starting at the public directory. For example, if there's a request for the file http://localhost/static/css/bootstrap.min.css the server will look for the file

```
<application root>/css/bootstrap.min.css
```

When it's found, the server will serve it as it is, without processing it first.

2.4.3 Creating the handler function

In a previous section you used `HandleFunc` to redirect the request to a handler function. Handler functions are nothing more than Go functions that take a `ResponseWriter` as the first parameter and a pointer to a `Request` as the second, shown next.

Listing 2.2 The `index` handler function in main.go

```
func index(w http.ResponseWriter, r *http.Request) {
  files := []string{"templates/layout.html",
                    "templates/navbar.html",
                    "templates/index.html",}
  templates := template.Must(template.ParseFiles(files...))
  threads, err := data.Threads(); if err == nil {
```

```
    templates.ExecuteTemplate(w, "layout", threads)
  }
}
```

Notice that you're using the `Template` struct from the html/template standard library so you need to add that in the list of imported libraries. The `index` handler function doesn't do anything except generate the HTML and write it to the `ResponseWriter`. We'll cover generating HTML in the upcoming section.

We've talked about handler functions that handle requests for the root URL (/), but there are a number of other handler functions. Let's look at the rest of them in the following listing, also in the main.go file.

Listing 2.3 ChitChat main.go source file

```
package main

import (
  "net/http"
)

func main() {

  mux := http.NewServeMux()
  files := http.FileServer(http.Dir(config.Static))
  mux.Handle("/static/", http.StripPrefix("/static/", files))

  mux.HandleFunc("/", index)
  mux.HandleFunc("/err", err)

  mux.HandleFunc("/login", login)
  mux.HandleFunc("/logout", logout)
  mux.HandleFunc("/signup", signup)
  mux.HandleFunc("/signup_account", signupAccount)
  mux.HandleFunc("/authenticate", authenticate)

  mux.HandleFunc("/thread/new", newThread)
  mux.HandleFunc("/thread/create", createThread)
  mux.HandleFunc("/thread/post", postThread)
  mux.HandleFunc("/thread/read", readThread)

  server := &http.Server{
    Addr:           "0.0.0.0:8080",
    Handler:        mux,
  }
  server.ListenAndServe()
}
```

You might notice that the various handler functions aren't defined in the same main.go file. Instead, I split the definition of the handler functions in other files (please refer to the code in the GitHub repository). So how do you link these files? Do you write code to include the other files like in PHP, Ruby, or Python? Or do you run a special command to link them during compile time?

In Go, you simply make every file in the same directory part of the `main` package and they'll be included. Alternatively, you can place them in a separate package and import them. We'll use this strategy when connecting with the database, as you'll see later.

2.4.4 Access control using cookies

As in many web applications, ChitChat has public pages that are available to anyone browsing to those pages, as well as private pages that require users to log into their account first.

Once the user logs in, you need to indicate in subsequent requests that the user has already logged in. To do this, you write a cookie to the response header, which goes back to the client and is saved at the browser. Let's look at the `authenticate` handler function, which authenticates the user and returns a cookie to the client. The `authenticate` handler function is in the route_auth.go file, shown next.

Listing 2.4 The `authenticate` handler function in route_auth.go

```
func authenticate(w http.ResponseWriter, r *http.Request) {
  r.ParseForm()
  user, _ := data.UserByEmail(r.PostFormValue("email"))
  if user.Password == data.Encrypt(r.PostFormValue("password")) {
    session := user.CreateSession()
    cookie := http.Cookie{
      Name:      "_cookie",
      Value:     session.Uuid,
      HttpOnly:  true,
    }
    http.SetCookie(w, &cookie)
    http.Redirect(w, r, "/", 302)
  } else {
    http.Redirect(w, r, "/login", 302)
  }
}
```

Note that in the source code in the previous listing that we haven't yet discussed `data.Encrypt` and `data.UserbyEmail`. In order to keep with the flow, I won't explain these functions in detail; their names make them self-explanatory. For example, `data.UserByEmail` retrieves a `User` struct given the email; `data.Encrypt` encrypts a given string. We'll get into the `data` package later in this chapter. For now let's return to the authentication handler flow.

First, you need to authenticate the user. You must make sure the user exists and the user's encrypted password in the database is the same as the encrypted password posted to the handler. Once the user is authenticated, you create a `Session` struct using `user.CreateSession`, a method on the `User` struct. `Session` looks like this:

```
type Session struct {
  Id        int
  Uuid      string
  Email     string
```

```
  UserId    int
  CreatedAt time.Time
}
```

The `Email` named field stores the email of the user who is logged in; the `UserId` named field contains the ID of the user table row with the user information. The most important information is the `Uuid`, which is a randomly generated unique ID. `Uuid` is the value you want to store at the browser. The session record itself is stored in the database.

Once you have the session record created, you create the `Cookie` struct:

```
cookie := http.Cookie{
  Name:      "_cookie",
  Value:     session.Uuid,
  HttpOnly:  true,
}
```

The name is arbitrary and the value is the unique data that's stored at the browser. You don't set the expiry date so that the cookie becomes a session cookie and it's automatically removed when the browser shuts down. You set `HttpOnly` to only allow HTTP or HTTPS to access the cookie (and not other non-HTTP APIs like JavaScript).

To add the cookie to the response header, use this code:

```
http.SetCookie(writer, &cookie)
```

Now that we have the cookie in the browser, you want to be able to check in the handler function whether or not the user is logged in. You create a utility function called `session` that you'll be able to reuse in other handler functions. The `session` function, shown in the next listing, and all other utility functions are written to the util.go file. Note that even though you placed the function in a separate file, it's still part of the `main` package, so you can use it directly without mentioning the package name, unlike in `data.Encrypt`.

Listing 2.5 `session` utility function in util.go

```
func session(w http.ResponseWriter, r *http.Request)(sess data.Session, err
   error){
  cookie, err := r.Cookie("_cookie")
  if err == nil {
    sess = data.Session{Uuid: cookie.Value}
    if ok, _ := sess.Check(); !ok {
      err = errors.New("Invalid session")
    }
  }
  return
}
```

The `session` function retrieves the cookie from the request:

```
cookie, err := r.Cookie("_cookie")
```

If the cookie doesn't exist, then obviously the user hasn't logged in yet. If it exists, the `session` function performs a second check and checks the database to see if the session's unique ID exists. It does this by using the `data.Session` function (that you'll create in a bit) to retrieve the session and then calling the `Check` method on that session:

```
sess = data.Session{Uuid: cookie.Value}
if ok, _ := sess.Check(); !ok {
  err = errors.New("Invalid session")
}
```

Now that you're able to check and differentiate between a user who has logged in and a user who hasn't, let's revisit our `index` handler function, shown in the following listing, and see how you can use this `session` function (code shown in bold).

Listing 2.6 The `index` handler function

```
func index(w http.ResponseWriter, r *http.Request) {
  threads, err := data.Threads(); if err == nil {
    _, err := session(w, r)
    public_tmpl_files := []string{"templates/layout.html",
                                  "templates/public.navbar.html",
                                  "templates/index.html"}
    private_tmpl_files := []string{"templates/layout.html",
                                   "templates/private.navbar.html",
                                   "templates/index.html"}
    var templates *template.Template
    if err != nil {
      templates = template.Must(template.Parse-
   Files(private_tmpl_files...))
    } else {
      templates = template.Must(template.ParseFiles(public_tmpl_files...))
    }
    templates.ExecuteTemplate(w, "layout", threads)
  }
}
```

The `session` function returns a `Session` struct, which you can use to extract user information, but we aren't interested in that right now, so assign it to the *blank identifier* (_). What we are interested in is `err`, which you can use to determine whether the user is logged in and specify that the `public` navigation bar or the `private` navigation bar should be shown.

That's all there is to it. We're done with the quick overview of processing requests; we'll get on with generating HTML for the client next, and continue where we left off earlier.

2.5 Generating HTML responses with templates

The logic in the `index` handler function was mainly about generating HTML for the client. Let's start by defining a list of template files that you'll be using in a Go slice (I'll show private_tmpl_files here; public_tmpl_files is exactly the same).

```
private_tmpl_files := []string{"templates/layout.html",
                               "templates/private.navbar.html",
                               "templates/index.html"}
```

The three files are HTML files with certain embedded commands, called *actions*, very similar to other template engines like Mustache or CTemplate. Actions are annotations added to the HTML between `{{` and `}}`.

You parse these template files and create a set of templates using the `ParseFiles` function. After parsing, you wrap the `Must` function around the results. This is to catch errors (the `Must` function panics when a `ParseFiles` returns an error).

```
templates := template.Must(template.ParseFiles(private_tmpl_files...))
```

We've talked a lot about these template files; let's look at them now.

Each template file defines a template (templates are described in detail in chapter 5). This is not mandatory—you don't need to define templates for every file—but doing so is useful, as you'll see later. In the layout.html template file, you begin with the `define` action, which indicates that the chunk of text starting with `{{ define "layout" }}` and ending with `{{ end }}` is part of the layout template, as shown next.

Listing 2.7 layout.html template file

```
{{ define "layout" }}

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=9">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>ChitChat</title>
    <link href="/static/css/bootstrap.min.css" rel="stylesheet">
    <link href="/static/css/font-awesome.min.css" rel="stylesheet">
  </head>
  <body>
    {{ template "navbar" . }}

    <div class="container">

      {{ template "content" . }}

    </div> <!-- /container -->

    <script src="/static/js/jquery-2.1.1.min.js"></script>
    <script src="/static/js/bootstrap.min.js"></script>
  </body>
</html>

{{ end }}
```

Within the layout template, we have two other actions, both of which indicate positions where another template can be included. The dot (.) that follows the name of

the template to be included is the data passed into the template. For example, listing 2.7 has `{{ template "navbar" . }}`, which indicates that the template named *navbar* should be included at that position, and the data passed into the layout template should be passed on to the navbar template too.

The navbar template in the public.navbar.html template file is shown next. The navbar template doesn't have any actions other than defining the template itself (actions aren't strictly necessary in template files).

Listing 2.8 navbar.html template file

```
{{ define "navbar" }}

<div class="navbar navbar-default navbar-static-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed"
      ➥ data-toggle="collapse" data-target=".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">
        <i class="fa fa-comments-o"></i>
        ChitChat
      </a>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li><a href="/">Home</a></li>
      </ul>
      <ul class="nav navbar-nav navbar-right">
        <li><a href="/login">Login</a></li>
      </ul>
    </div>
  </div>
</div>

{{ end }}
```

Let's look at the content template in last template file, index.html, in the following listing. Notice that the name of the template doesn't necessary need to match the name of the template file, even though that has been the case for the past two files.

Listing 2.9 index.html template

```
{{ define "content" }}

<p class="lead">
  <a href="/thread/new">Start a thread</a> or join one below!
</p>
```

```
{{ range . }}
  <div class="panel panel-default">
    <div class="panel-heading">
      <span class="lead"> <i class="fa fa-comment-o"></i> {{ .Topic }}</span>
    </div>
    <div class="panel-body">
      Started by {{ .User.Name }} - {{ .CreatedAtDate }} - {{ .NumReplies }}
   posts.
      <div class="pull-right">
        <a href="/thread/read?id={{.Uuid }}">Read more</a>
      </div>
    </div>
  </div>
{{ end }}

{{ end }}
```

The code in index.html is interesting. You'll notice a number of actions within the content template that start with a dot (`.`), such as `{{ .User.Name }}` and `{{ .CreatedAtDate }}`. To understand where this comes from, we need to go back to the `index` handler function.

```
  threads, err := data.Threads(); if err == nil {
    templates.ExecuteTemplate(writer, "layout", threads)
  }
```

Let's start off with this:

```
templates.ExecuteTemplate(writer, "layout", threads)
```

We take the set of templates we parsed earlier, and execute the layout template using `ExecuteTemplate`. Executing the template means we take the content from the template files, combine it with data from another source, and generate the final HTML content, shown in figure 2.6.

Why the layout template and not the other two templates? This should be obvious: the layout template includes the other two templates, so if we execute the layout template, the other two templates will also be executed and the intended HTML will be

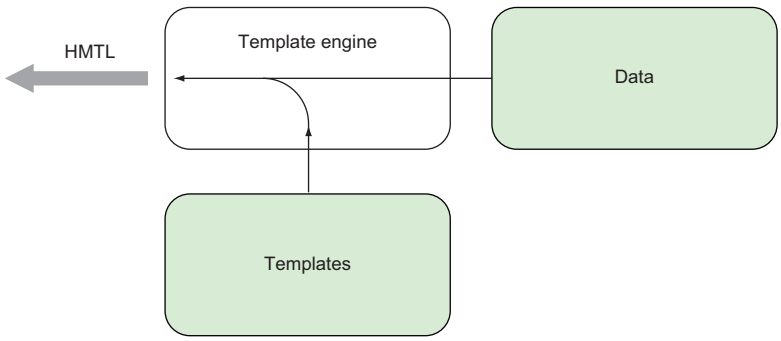

Figure 2.6 The template engine combines the data and template to produce HTML.

generated. If we executed either one of the other two templates, we would only get part of the HTML we want.

As you might realize by now, the dot (.) represents the data that's passed into the template (and a bit more, which is explained in the next section). Figure 2.7 shows what we end up with.

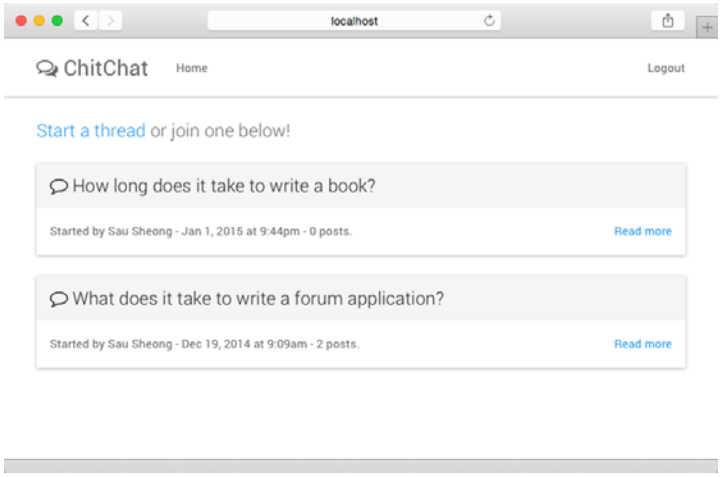

Figure 2.7 The index page of the example ChitChat web application

2.5.1 Tidying up

HTML generation will be used over and over again, so let's do some tidying up and move those steps into a function called `generateHTML`, shown next.

Listing 2.10 The `generateHTML` function

```
func generateHTML(w http.ResponseWriter, data interface{}, fn ...string) {
  var files []string
  for _, file := range fn {
    files = append(files, fmt.Sprintf("templates/%s.html", file))
  }
  templates := template.Must(template.ParseFiles(files...))
  templates.ExecuteTemplate(writer, "layout", data)
}
```

`generateHTML` takes a `ResponseWriter`, some data, and a list of template files to be parsed. The `data` parameter is the empty interface type, which means it can take in any type. This might come as a surprise if you're a new Go programmer; isn't Go a statically typed programming language? What's this about accepting any types in as a parameter?

As it turns out, Go has an interesting way of getting around being a statically typed programming language and it provides the flexibility of accepting different types, using interfaces. Interfaces in Go are constructs that are sets of methods and are also

types. An empty interface is then an empty set, meaning any type can be an empty interface; you can pass any type into this function as the data.

The last parameter in the function starts with … (three dots). This indicates that the `generateHTML` function is a *variadic* function, meaning it can take zero or more parameters in that last variadic parameter. This allows you to pass any number of template files to the function. Variadic parameters need to be the last parameter for the variadic function.

Now that we have the `generateHTML` function, let's go back and clean up the `index` handler function. The new `index` handler function, shown here, now looks a lot neater.

Listing 2.11 The final `index` handler function

```
func index(writer http.ResponseWriter, request *http.Request) {
  threads, err := data.Threads(); if err == nil {
    _, err := session(writer, request)
    if err != nil {
      generateHTML(writer, threads, "layout", "public.navbar", "index")
    } else {
      generateHTML(writer, threads, "layout", "private.navbar", "index")
    }
  }
}
```

We sort of glossed over the data source and what we used to combine with the templates to get the final HTML. Let's get to that now.

2.6 Installing PostgreSQL

In this chapter as well as for any remaining chapters in the book that require access to a relational database, we'll be using PostgreSQL. Before we start any code, I'll run through how to install and start up PostgreSQL, and also create the database that we need for this chapter.

2.6.1 Linux/FreeBSD

Prebuilt binaries are available for many variants of Linux and FreeBSD from www.postgresql.org/download. Download any one of them from the site and follow the instructions. For example, you can install Postgres on Ubuntu by executing this command on the console:

```
sudo apt-get install postgresql postgresql-contrib
```

This will install both the `postgres` package and an additional package of utilities, and also start it up.

By default Postgres creates a `postgres` user and that's the only user who can connect to the server. For convenience you can create another Postgres account with your username. First, you need to log in to the Postgres account:

```
sudo su postgres
```

Next, use `createuser` to create your postgreSQL account:

```
createuser -interactive
```

Finally, use `createdb` to create your database:

```
createdb  <YOUR ACCOUNT NAME>
```

2.6.2 Mac OS X

One of the easiest ways to install PostgreSQL on Mac OS X is to use the Postgres application. Download the zip file and unpack it. Then drag and drop the Postgres.app file into your Applications folder and you're done. You can start the application just like you start any Mac OS X application. The first time you start the application, Postgres will initialize a new database cluster and create a database for you. The command-line tool psql is part of the package, so you'll be able to access the database using psql once you set the correct path. Open up Terminal and add this line your ~/.profile or ~/.bashrc file:

```
export PATH=$PATH:/Applications/Postgres.app/Contents/Versions/9.4/bin
```

2.6.3 Windows

Installing PostgreSQL on Windows is fairly straightforward too. There are a number of graphical installers on Windows that do all the heavy lifting for you; you simply need to provide the settings accordingly. A popular installer is one from Enterprise DB at www.enterprisedb.com/products-services-training/pgdownload.

A number of tools, including pgAdmin III, are installed along with the package, which allows you to set up the rest of the configuration.

2.7 Interfacing with the database

In the design section earlier in this chapter, we talked about the four data structures used in ChitChat. Although you can place the data structures in the same main file, it's neater if you store all data-related code in another package, aptly named `data`.

To create a package, create a subdirectory called data and create a file named thread.go to store all thread-related code (you'll create a user.go file to store all user-related code). Then, whenever you need to use the `data` package (for example, in the handlers that need to access the database), you import the package:

```
import (
  "github.com/sausheong/gwp/Chapter_2_Go_ChitChat/chitchat/data"
)
```

Within the thread.go file, define a `Thread` struct, shown in the following listing, to contain the data.

Listing 2.12 The `Thread` struct

```
package data

import(
  "time"
)
```

```
type Thread struct {
  Id        int
  Uuid      string
  Topic     string
  UserId    int
  CreatedAt time.Time
}
```

Notice that the package name is no longer `main` but `data` (in bold). When you use anything in this package later (functions or structs or anything else), you need to provide the package name along with it. If you want to use the `Thread` struct you must use `data.Thread` instead of just `Thread` alone. This is the `data` package you used earlier in the chapter. Besides containing the structs and code that interact with the database, the package contains other functions that are closely associated.

The `Thread` struct should correspond to the DDL (Data Definition Language, the subset of SQL) that's used to create the relational database table called `threads`. You don't have these tables yet so let's create them first. Of course, before you create the database tables, you should create the database itself. Let's create a database called `chitchat`. Execute this command at the console:

```
createdb chitchat
```

Once you have the database, you can use setup.sql to create the database tables for ChitChat, shown next.

Listing 2.13 setup.sql used to create database tables in PostgreSQL

```
create table users (
  id         serial primary key,
  uuid       varchar(64) not null unique,
  name       varchar(255),
  email      varchar(255) not null unique,
  password   varchar(255) not null,
  created_at timestamp not null
);

create table sessions (
  id         serial primary key,
  uuid       varchar(64) not null unique,
  email      varchar(255),
  user_id    integer references users(id),
  created_at timestamp not null
);

create table threads (
  id         serial primary key,
  uuid       varchar(64) not null unique,
  topic      text,
  user_id    integer references users(id),
  created_at timestamp not null
);

create table posts (
  id         serial primary key,
```

```
  uuid       varchar(64) not null unique,
  body       text,
  user_id    integer references users(id),
  thread_id  integer references threads(id),
  created_at timestamp not null
);
```

To run the script, use the psql tool that's usually installed as part of your PostgreSQL installation (see the previous section). Go to the console and run this command:

```
psql -f setup.sql -d chitchat
```

This command should create the necessary database tables in your database. Once you have your database tables, you must be able to connect to the database and do stuff with the tables. So you'll create a global variable, `Db`, which is a pointer to `sql.DB`, a representation of a pool of database connections. You'll define `Db` in the data.go file, as shown in the following listing. Note that this listing also contains a function named `init` that initializes `Db` upon startup of your web application. You'll use `Db` to execute your queries.

Listing 2.14 The `Db` global variable and the `init` function in data.go

```
Var Db *sql.DB

func init() {
  var err error
  Db, err = sql.Open("postgres", "dbname=chitchat sslmode=disable")
  if err != nil {
    log.Fatal(err)
  }
  return
}
```

Now that you have the struct, the tables, and a database connection pool, how do you connect the `Thread` struct with the `threads` table? There's no particular magic to it. As with everything else in ChitChat, you simply create a function every time you want interaction between the struct and the database. To extract all threads in the database for the `index` handler function, create a `Threads` function in thread.go, as shown next.

Listing 2.15 The `Threads` function in thread.go

```
func Threads() (threads []Thread, err error){
  rows, err := Db.Query("SELECT id, uuid, topic, user_id, created_at FROM
   threads ORDER BY created_at DESC")
  if err != nil {
    return
  }
  for rows.Next() {
    th := Thread{}
    if err = rows.Scan(&th.Id, &th.Uuid, &th.Topic, &th.UserId,
    ➥ &th.CreatedAt); err != nil {
```

```
      return
    }
    threads = append(threads, th)
  }
  rows.Close()
  return
}
```

Without getting into the details (which will be covered in chapter 6), these are the general steps:

1. Connect to the database using the database connection pool.
2. Send an SQL query to the database, which will return one or more rows.
3. Create a struct.
4. Iterate through the rows and scan them into the struct.

In the `Threads` function, you return a slice of the `Thread` struct, so you need to create the slice and then continually append to it until you're done with all the rows.

Now that you can get the data from the database into the struct, how do you get the data in the struct to the templates? Let's return to the index.html template file (listing 2.9), where you find this code:

```
{{ range . }}
  <div class="panel panel-default">
    <div class="panel-heading">
      <span class="lead"> <i class="fa fa-comment-o"></i> {{ .Topic }}</span>
    </div>
    <div class="panel-body">
      Started by {{ .User.Name }} - {{ .CreatedAtDate }} - {{ .NumReplies }}
   posts.
      <div class="pull-right">
        <a href="/thread/read?id={{.Uuid }}">Read more</a>
      </div>
    </div>
  </div>
{{ end }}
```

As you'll recall, a dot (.) in an action represents the data that's passed into the template to be combined to generate the final output. The dot here, as part of `{{ range . }}`, is the `threads` variable extracted earlier using the `Threads` function, which is a slice of `Thread` structs.

The `range` action assumes that the data passed in is either a slice or an array of structs. The `range` action allows you to iterate through and access the structs using their named fields. For example, `{{ .Topic }}` allows you to access the `Topic` field of the `Thread` struct. Note that the field must start with a dot and the name of the field is capitalized.

What about `{{ .User.Name }}` and `{{ .CreatedAtDate }}` and `{{ .NumReplies }}`? The `Thread` struct doesn't have these as named fields, so where do they come from? Let's look at `{{ .NumReplies }}`. While using the name of a field after the dot accesses the data in the struct, you can do the same with a special type of function called *methods*.

Methods are functions that are attached to any named types (except a pointer or an interface), including structs. By attaching a function to a pointer to a `Thread` struct, you allow the function to access the thread. The `Thread` struct, also called the *receiver*, is normally changed after calling the method.

The `NumReplies` method is shown here.

Listing 2.16 `NumReplies` method in thread.go

```
func (thread *Thread) NumReplies() (count int) {
  rows, err := Db.Query("SELECT count(*) FROM posts where thread_id = $1",
   thread.Id)
  if err != nil {
    return
  }
  for rows.Next() {
    if err = rows.Scan(&count); err != nil {
      return
    }
  }
  rows.Close()
  return
}
```

The `NumReplies` method opens a connection to the database, gets the count of threads using an SQL query, and scans it into the `count` parameter passed into the method. The `NumReplies` method returns this count, which is then used to replace `.NumReplies` in the HTML, by the template engine, shown in figure 2.8.

By providing a combination of functions and methods on the data structs (`User`, `Session`, `Thread`, and `Post`), you create a data layer that shields you from directly accessing the database in the handler functions. Although there are plenty of libraries that provide this functionality, it's good to understand that the underlying basis of accessing the database is quite easy, with no magic involved. Just simple, straightforward code.

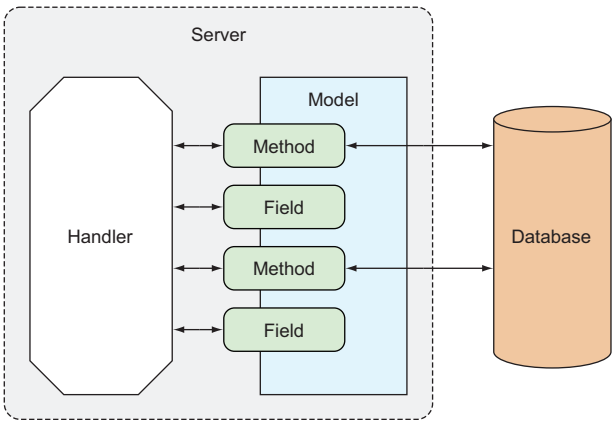

Figure 2.8 Connecting the struct model with the database and the handler

2.8 Starting the server

Let's round out this chapter by showing code that starts up the server and attaches the multiplexer to the server. This is part of the `main` function, so it will be in main.go.

```
server := &http.Server{
  Addr:     "0.0.0.0:8080",
  Handler:  mux,
}
server.ListenAndServe()
```

The code is simple; you create a `Server` struct and call the `ListenAndServe` function on it and you get your server.

Now let's get it up and running. Compile this from the console:

```
go build
```

This command will create a binary executable file named chitchat in the same directory (and also in in your $GOPATH/bin directory). This is our ChitChat server. Let's start the server:

```
./chitchat
```

This command will start the server. Assuming that you've created the necessary database tables, go to http://localhost:8080 and registered for an account; then log in and start creating your own forum threads.

2.9 Wrapping up

We went through a 20,000-foot overview of the various building blocks of a Go web application. Figure 2.9 shows a final recap of the entire flow. As illustrated,

1. The client sends a request to the server.
2. This is received by the multiplexer, which redirects it to the correct handler.

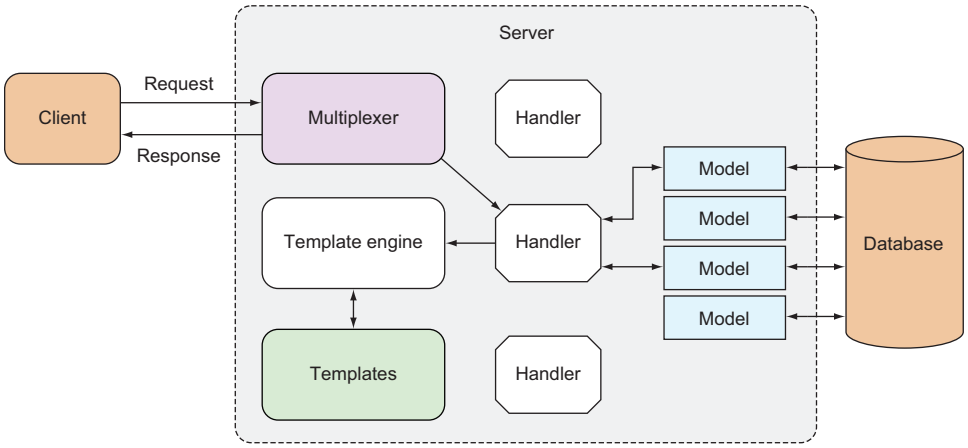

Figure 2.9 The web application big picture

3. The handler processes the request.
4. When data is needed, it will use one or more data structs that model the data in the database.
5. The model connects with the database, triggered by functions or methods on the data struct.
6. When processing is complete, the handler triggers the template engine, sometimes sending in data from the model.
7. The template engine parses template files to create templates, which in turn are combined with data to produce HTML.
8. The generated HTML is sent back to the client as part of the response.

And we're done! In the next few chapters, we will dive in deeper into this flow and get into the details of each component.

2.10 Summary

- Receiving and processing requests are the heart of any web application.
- The multiplexer redirects HTTP requests to the correct handler for processing, including static files.
- Handler functions are nothing more than Go functions that take a `ResponseWriter` as the first parameter and a pointer to a `Request` as the second.
- Cookies can be used as a mechanism for access control.
- HTML responses can be generated by parsing template files together with data to provide the final HTML data that is returned to the calling browser.
- Persisting data to a relational database can be done through direct SQL using the `sql` package.

GO/PROGRAMMING

Go Web Programming

Sau Sheong Chang

Free eBook
SEE INSERT

The Go language handles the demands of scalable, high-performance web applications by providing clean and fast compiled code, garbage collection, a simple concurrency model, and a fantastic standard library. It's perfect for writing microservices or building scalable, maintainable systems.

Go Web Programming teaches you how to build web applications in Go using modern design principles. You'll learn how to implement the dependency injection design pattern for writing test doubles, use concurrency in web applications, and create and consume JSON and XML in web services. Along the way, you'll discover how to minimize your dependence on external frameworks, and you'll pick up valuable productivity techniques for testing and deploying your applications.

What's Inside

- Basics
- Testing and benchmarking
- Using concurrency
- Deploying to standalone servers, PaaS, and Docker
- Dozens of tips, tricks, and techniques

This book assumes you're familiar with Go language basics and the general concepts of web development.

Sau Sheong Chang is Managing Director of Digital Technology at Singapore Power and an active contributor to the Ruby and Go communities.

"As the importance of the Go language grows, the need for a great tutorial grows with it. This book fills this need."
—Shaun Lippy
Oracle Corporation

"An excellent book, whether you are an experienced gopher, or you know web development but are new to Go."
—Benoit Benedetti
University of Nice

"Everything you need to get started writing and deploying web apps in Go."
—Brian Cooksey, Zapier

"Definitive how-to guide for web development in Go."
—Gualtiero Testa, Factor-y S.r.l.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/go-web-programming

ISBN-13: 978-1-61729-256-9
ISBN-10: 1-61729-256-7
54499
9 781617 292569

MANNING

$44.99 / Can $51.99 [INCLUDING eBOOK]