```
package main

import (
    "bytes"
    "encoding/json"
    "errors"
    "fmt"
    "html/template"
    "net/http"
    "path/filepath"
    "strconv"

    "github.com/gbih/snippetbox/pkg/models"
)

func (nfs neuteredFileSystem) Open(path string) (http.File, error) {
    f, err := nfs.fs.Open(path)
    if err != nil {
        return nil, err
    }

    s, err := f.Stat()
    if s.IsDir() {
        index := filepath.Join(path, "index.html")
        if _, err := nfs.fs.Open(index); err != nil {
            closeErr := f.Close()
            if closeErr != nil {
                return nil, closeErr
            }

            return nil, err
        }
    }

    return f, nil
}

//----------

// API Usage: curl -i localhost:4000/api/v1/test

func (app *application) apiTest(w http.ResponseWriter, r *http.Request) {
    data := []struct {
        Name     string ‘json:"name"‘
        Location string ‘json:"location"‘
    }{
        {"George", "æ\227¥æ\234¬"},
        {"Izumi", "è¶\212å¾\214æ¹¯æ²¢"},
    }

    js, err := json.MarshalIndent(data, "", "\t")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    w.Header().Set("Content-Type", "application/json; charset=utf-8")
    w.Write(js)
}

//----------

// Example to show explicit code flow

func (app *application) homeOriginal(w http.ResponseWriter, r *http.Request) {

    s, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
```

```
        return
    }

    files := []string{
        "./ui/html/home.pageOriginal.html",
        "./ui/html/base.layout.html",
        "./ui/html/footer.partial.html",
    }

    // Create a FuncMap with which to register the function.
    funcMap := template.FuncMap{
        // "humanDate" is the function name created in templates.go,
        //  to be called in the template text.
        "humanDate": humanDate,
    }

    // Create a template, add the function map, and parse the text.
    ts, err := template.New("base").Funcs(funcMap).ParseFiles(files...)
    if err != nil {
        app.serverError(w, err)
        return
    }

    // As soon as we begin adding dynamic behavior to our HTML templates
    // thereâ\200\231s a risk of encountering runtime errors. We want to avoid
    // the use-case of the template compiling ok, but throwing a run-time error.
    // Essentially, To fix this we need to make the template render a two-stage
    // process. First, we should make a trial-render by writing the template
    // into a buffer. If this fails, we can respond to the user with an error
    // message. But if it works, we can then write the contents of the buffer
    // to our http.ResponseWriter.

    // Initialize a new buffer.
    buf := new(bytes.Buffer)

    // Write the template to the buffer, instead of straight to the
    // http.ResponseWriter. If there’s an error, call our serverError helper
    // and then return.
    err = ts.Execute(buf, &templateData{
        Snippets: s,
    })
    if err != nil {
        app.serverError(w, err)
        return
    }

    // Write the contents of the buffer to the http.ResponseWriter. Again, this
    // is another time where we pass our http.ResponseWriter to a function that
    // takes an io.Writer.
    buf.WriteTo(w)

    // OLD:
    // err = ts.Execute(w, &templateData{
    //  Snippets: s,
    // })
    // if err != nil {
    //  app.serverError(w, err)
    // }

}

//----------

func (app *application) home(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
        app.notFound(w)
        return
    }
```

```
s, err := app.snippets.Latest()
if err != nil {
    app.serverError(w, err)
    return
}

// New code
app.render(w, r, "home.page.html", &templateData{
    Snippets: s,
})

/*
    Note:
    1.  Old template parsing and execution code refactored out, moved to
    templates.go in this function:
    newTemplateCache(dir string) (map[string]*template.Template, error)

    2. This cache is initialized in the main() function and made
    available to handlers as a dependency via the application struct.

        type application struct {
            templateCache map[string]*template.Template
        }

    and added as to the application dependencies
        app := &application{
            templateCache: templateCache,
        }

    So now we have an in-memory cache of the relevant template set for
    each of our pages, and our handlers have access to this cache via
    the application struct.

    Now, create a helper render method so that we can easily render
    the templates from the cache.
    File: cmd/web/helpers.go

    func (app *application) render(
            w http.ResponseWriter,
            r *http.Request,
            name string,
            td *templateData) { ... }

    Now, we can access these cached templates via the render helper function:
    Use the new render helper.
    app.render(w, r, "home.page.tmpl", &templateData{
        Snippets: s,
    })
*/
/*
    Old code:
    data := &templateData{Snippets: s}

    files := []string{
        "./ui/html/home.page.html",
        "./ui/html/base.layout.html",
        "./ui/html/footer.partial.html",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        app.serverError(w, err)
        return
    }

    err = ts.Execute(w, data)
    if err != nil {
```

```
            app.serverError(w, err)
        }
    */
}

// API Usage : curl -i localhost:4000/api/v1/home

func (app *application) apiHome(w http.ResponseWriter, r *http.Request) {

    s, err := app.snippets.Latest()
    if err != nil {
        app.serverError(w, err)
        return
    }

    js, err := json.MarshalIndent(s, "", "\t")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.Header().Set("Content-Type", "application-json; charset=UTF-8")
    w.Write(js)
}

//----------

func (app *application) showSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get("id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            app.notFound(w)
        } else {
            app.serverError(w, err)
        }
        return
    }

    // Manual template parsing and execution code refactored out

    app.render(w, r, "show.page.html", &templateData{
        Snippet: s,
    })

}

// API Usage: curl -i 'http://localhost:4000/api/v1/snippet?id=1'

func (app *application) apiShowSnippet(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.URL.Query().Get("id"))
    if err != nil || id < 1 {
        app.notFound(w)
        return
    }

    s, err := app.snippets.Get(id)
    if err != nil {
        if errors.Is(err, models.ErrNoRecord) {
            app.notFound(w)
        } else {
            app.serverError(w, err)
        }
```

```
        return
    }
    js, err := json.MarshalIndent(s, "", "\t")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    //fmt.Fprintf(w, "%v", s)
    w.Header().Set("Content-Type", "application-json; charset=utf-8")
    w.Write(js)
}

//----------

func (app *application) createSnippet(w http.ResponseWriter, r *http.Request) {
    if r.Method != http.MethodPost {
        w.Header().Set("Allow", http.MethodPost)
        http.Error(w, "Method Not Allowed", 405)
        return
    }

    title := "O snail"
    content := "O snail\nBut slowly, slowly!\n\nâ\200\223 Kobayashi Issa"
    expires := "7"

    id, err := app.snippets.Insert(title, content, expires)
    if err != nil {
        app.serverError(w, err)
        return
    }

    http.Redirect(w, r, fmt.Sprintf("/snippet?id=%d", id), http.StatusSeeOther)

}
```

```
package main

import (
    "bytes"
    "fmt"
    "net/http"
    "runtime/debug"
    "time"
)

// Create an addDefaultData helper. This takes a pointer to a templateData
// struct, adds the current year to the CurrentYear field, and then returns
// the pointer. Again, we're not using the *http.Request parameter at the
// moment, but we will do later in the book.
func (app *application) addDefaultData(td *templateData, r *http.Request) *templateData {
    if td == nil {
        td = &templateData{}
    }
    td.CurrentYear = time.Now().Year()
    return td
}

func (app *application) serverError(w http.ResponseWriter, err error) {
    trace := fmt.Sprintf("%s\n%s", err.Error(), debug.Stack())
    app.errorLog.Println(trace)
    http.Error(w, http.StatusText(http.StatusInternalServerError),
http.StatusInternalServerError)
}

func (app *application) clientError(w http.ResponseWriter, status int) {
    http.Error(w, http.StatusText(status), status)
}

func (app *application) notFound(w http.ResponseWriter) {
    app.clientError(w, http.StatusNotFound)
}

func (app *application) render(w http.ResponseWriter, r *http.Request, name string, td
*templateData) {
    ts, ok := app.templateCache[name]
    if !ok {
        app.serverError(w, fmt.Errorf("The template %s does not exist", name))
        return
    }

    // Need to make the template render a two-stage process. First, we should make a
â\200\230trialâ\200\231 render by writing the template into a buffer. If this fails, we
can respond to the user with an error message. But if it works, we can then write the
contents of the buffer to our http.ResponseWriter.

    // Initialize a new buffer.
    buf := new(bytes.Buffer)

    // Write the template to the buffer, instead of straight to the
    // http.ResponseWriter. If there's an error, call our serverError helper and then
    // return.
    //
    // Execute the template set, passing the dynamic data with the current
    // year injected.
    //err := ts.Execute(buf, td)
    err := ts.Execute(buf, app.addDefaultData(td, r))
    if err != nil {
        app.serverError(w, err)
        return
    }

    // Write the contents of the buffer to the http.ResponseWriter. Again, this
    // is another time where we pass our http.ResponseWriter to a function that
```

```
    // takes an io.Writer.
    buf.WriteTo(w)

}
```

```
package main

import (
    "database/sql"
    "flag"
    "fmt"
    "html/template"
    "log"
    "net/http"
    "os"

    "github.com/gbih/snippetbox/pkg/models/postgres"
    _ "github.com/lib/pq"
)

type neuteredFileSystem struct {
    fs http.FileSystem
}

type Config struct {
    Addr      string
    StaticDir string
}

type application struct {
    errorLog      *log.Logger
    infoLog       *log.Logger
    snippets      *postgres.SnippetModel
    templateCache map[string]*template.Template
}

func openDB(dsn string) (*sql.DB, error) {
    db, err := sql.Open("postgres", dsn)
    if err != nil {
        return nil, err
    }
    if err = db.Ping(); err != nil {
        return nil, err
    }
    return db, nil
}

func main() {

    cfg := new(Config)
    flag.StringVar(&cfg.Addr, "addr", ":4000", "HTTP network address")

    dsn := flag.String("dsn",
"postgres://postgres:postgres@localhost/snippetbox?sslmode=disable", "Postgres data
source name")

    flag.Parse()

    infoLog := log.New(os.Stdout, "INFO\t", log.Ldate|log.Ltime)
    errorLog := log.New(os.Stderr, "ERROR\t", log.Ldate|log.Ltime|log.Lshortfile)

    db, err := openDB(*dsn)
    if err != nil {
        errorLog.Fatal(err)
    }
    defer db.Close()

    templateCache, err := newTemplateCache("./ui/html/")
    if err != nil {
        errorLog.Fatal(err)
    }

    app := &application{
```

```
        errorLog:      errorLog,
        infoLog:       infoLog,
        snippets:      &postgres.SnippetModel{DB: db},
        templateCache: templateCache,
    }

    fmt.Println("**** templateCache: ", app.templateCache)

    srv := &http.Server{
        Addr:     cfg.Addr,
        ErrorLog: errorLog,
        Handler:  app.routes(),
    }

    infoLog.Printf("http://localhost%v/original", cfg.Addr)
    infoLog.Printf("http://localhost%v", cfg.Addr)
    infoLog.Printf("curl 'http://localhost%v/api/v1/test'", cfg.Addr)
    infoLog.Printf("curl 'http://localhost%v/api/v1/snippet?id=3'", cfg.Addr)

    err = srv.ListenAndServe()
    errorLog.Fatal(err)

}

// go run ./cmd/web >> ./logs/info.log 2>> ./logs/error.log
```

```
package main

import (
    "net/http"
)

func (app *application) routes() *http.ServeMux {

    mux := http.NewServeMux()
    mux.HandleFunc("/", app.home)
    mux.HandleFunc("/original", app.homeOriginal)
    mux.HandleFunc("/snippet", app.showSnippet)
    mux.HandleFunc("/snippet/create", app.createSnippet)

    fileServer := http.FileServer(neuteredFileSystem{http.Dir("./ui/static/")})
    mux.Handle("/static/", http.StripPrefix("/static", fileServer))

    // APIs
    mux.HandleFunc("/api/v1/snippet", app.apiShowSnippet)
    mux.HandleFunc("/api/v1/test", app.apiTest)
    mux.HandleFunc("/api/v1/home", app.apiHome)

    return mux
}
```

```
package main

import (
    "html/template"
    "path/filepath"
    "time"

    "github.com/gbih/snippetbox/pkg/models"
)

type templateData struct {
    CurrentYear int
    Snippet     *models.Snippet
    Snippets    []*models.Snippet
}

func humanDate(t time.Time) string {
    return t.Format("02 Jan 2006 at 15:04")
}

var functions = template.FuncMap{
    "humanDate": humanDate,
}

func newTemplateCache(dir string) (map[string]*template.Template, error) {

    cache := map[string]*template.Template{}

    pages, err := filepath.Glob(filepath.Join(dir, "*.page.html"))
    if err != nil {
        return nil, err
    }

    for _, page := range pages {
        name := filepath.Base(page)

        // The template.FuncMap must be registered with the template set before you
        // call the ParseFiles() method. This means we have to use template.New() to
        // create an empty template set, use the Funcs() method to register the
        // template.FuncMap, and then parse the file as normal.
        //ts, err := template.ParseFiles(page)
        ts, err := template.New(name).Funcs(functions).ParseFiles(page)
        if err != nil {
            return nil, err
        }

        ts, err = ts.ParseGlob(filepath.Join(dir, "*.layout.html"))
        if err != nil {
            return nil, err
        }

        ts, err = ts.ParseGlob(filepath.Join(dir, "*.partial.html"))
        if err != nil {
            return nil, err
        }

        cache[name] = ts
    }

    return cache, nil
}
```

```
package models

import (
    "errors"
    "time"
)

var ErrNoRecord = errors.New("models: no matching record found")

type Snippet struct {
    ID      int
    Title   string
    Content string
    Created time.Time
    Expires time.Time
}
```

```
package postgres

import (
    "database/sql"
    "errors"

    "github.com/gbih/snippetbox/pkg/models"
)

type SnippetModel struct {
    DB *sql.DB
}

func (m *SnippetModel) Insert(title, content, expires string) (int, error) {

    var id int

    stmt := 'INSERT INTO snippets
    (title, content, created, expires)
    VALUES
    ($1, $2, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP + INTERVAL '1 DAY' * $3)
    RETURNING id'

    err := m.DB.QueryRow(stmt, title, content, expires).Scan(&id)

    if err != nil {
        return 0, err
    }

    return int(id), nil

}

func (m *SnippetModel) Get(id int) (*models.Snippet, error) {

    stmt := 'SELECT id, title, content, created, expires FROM snippets
    WHERE expires > CURRENT_TIMESTAMP AND id = $1'

    row := m.DB.QueryRow(stmt, id)
    s := &models.Snippet{}

    err := row.Scan(&s.ID, &s.Title, &s.Content, &s.Created, &s.Expires)
    if err != nil {
        if errors.Is(err, sql.ErrNoRows) {
            return nil, models.ErrNoRecord
        } else {
            return nil, err
        }
    }

    return s, nil
}

func (m *SnippetModel) Latest() ([]*models.Snippet, error) {
    stmt := 'SELECT id, title, content, created, expires FROM snippets
    WHERE expires > CURRENT_TIMESTAMP ORDER BY created DESC LIMIT 3'

    rows, err := m.DB.Query(stmt)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    snippets := []*models.Snippet{}

    for rows.Next() {
        s := &models.Snippet{}
```

```
        err := rows.Scan(&s.ID, &s.Title, &s.Content, &s.Created, &s.Expires)
        if err != nil {
            return nil, err
        }

        snippets = append(snippets, s)
    }

    if err = rows.Err(); err != nil {
        return nil, err
    }

    return snippets, nil
}
```