Beginners Guide to Context

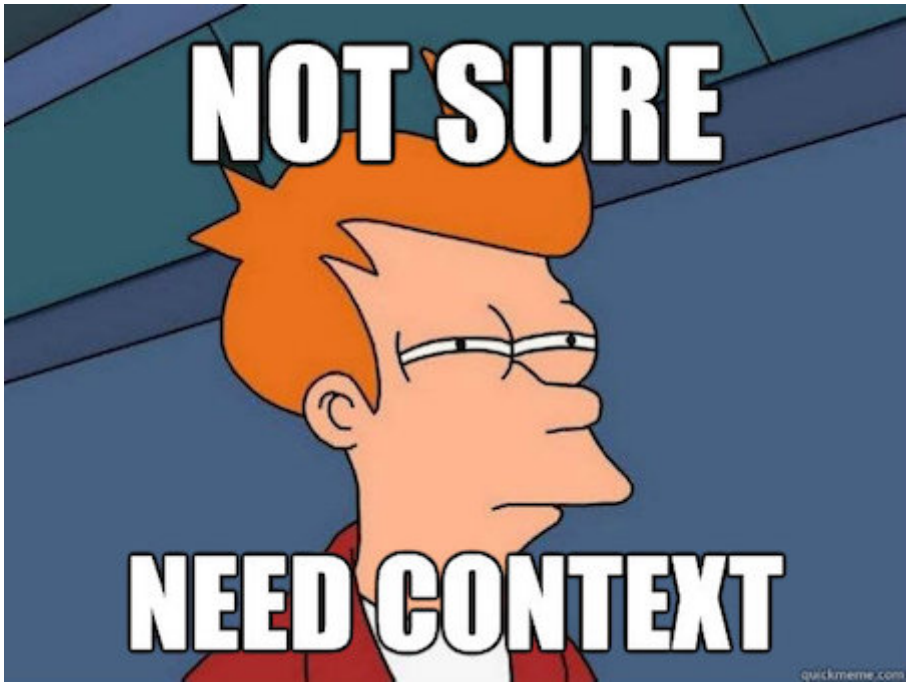
18 August 2016

Hi I'm Paul

- CTO @ Daily Burn
- Started using Go about 5 years ago
- We now use Go for queueing, real time messaging, ETL, devops

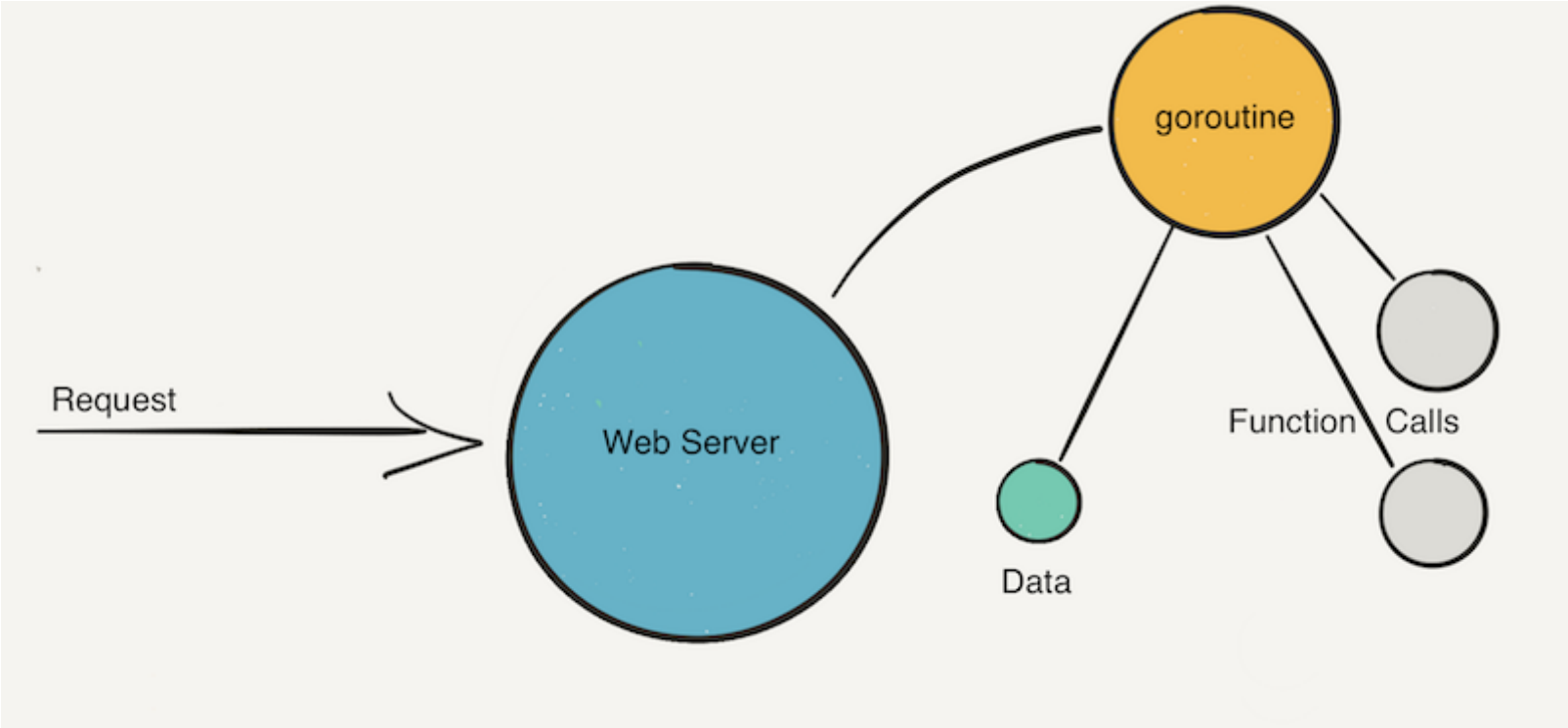
Let's Talk

- Goal of this talk is to introduce you to the context package
- How to use it, Where to use it
- Some best practices picked up from around the web



The Problem(s)

- In Go servers each new request spawns it's own goroutine
- Goroutines don't have any 'thread local' state
- Your code is responsible for things like things like cancellation, time outs and data



The Solution

- The **context** package provides a standard way to solve the problems of managing state during a request

context addresses:

- Request scoped data
- Cancellation, Deadlines & Timeouts
- It is safe for concurrent use

See: Cancellation, Context, and Plumbing by Sameer Ajmani (*GothamGo 2014*)

Some Context for context

- The **context** package originated out of Google and was announced officially in July 2014
- The package satisfies the need for request scoped data and provides a standardized way to handle cancellation and deadlines
- It provides a way to facilitate across API boundaries to goroutines created when handling a request
- For reference:

https://blog.golang.org/context (https://blog.golang.org/context)

https://blog.golang.org/pipelines (https://blog.golang.org/pipelines)

golang.org/x/net/context (https://godoc.org/golang.org/x/net/context)

Context and Go 1.7

- With the release of Go 1.7 context is now part of the core library
- The **context** package has been around long enough to have proven its worth
- Along with this are some additional changes to **net**, **net/http** and **os/exec**
- All of this will make it even easier to work with and are a great reason you should all consider using it in your projects
- **golang.org/x/net/context** becomes **context**

The Context Type

- **context** is made up of the Context Type along with some supporting functions

```
type Context interface {
    // Done returns a channel that is closed when this Context is canceled
    // or times out.
    Done() <-chan struct{}

    // Err indicates why this context was canceled, after the Done channel
    // is closed.
    Err() error

    // Deadline returns the time when this Context will be canceled, if any.
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    Value(key interface{}) interface{}
}
```

Done()

- The **Done** function returns a channel that acts as a cancellation signal to functions running on behalf of a context
- When the channel is closed the functions should end execution and exit

```
func someHandler() {
    ctx, cancel := context.WithCancel(context.Background())
    go doStuff(ctx)
    // ...some work happens...
    if someCondition {
        cancel()
    }
}

func doStuff(ctx context.Context) {
    // ...Doing some work
    select {
    case <-ctx.Done():
        fmt.Println("Stop work!")
        return
    }
}
```

Err(), Deadline()

- The Err() function returns an error indicating why the Context was cancelled
- The Deadline() function allows a sub-operation to determine if it should start work
- Deadline() returns both a time value indicating when work should be cancelled along with a boolean indicating if a deadline has been set on the context

```
func someHandler() {
        ctx, cancel := context.WithDeadline(context.Background(), time.Now().Add(5 * time.Second))
        go doStuff(ctx)
        // if deadline expires before work completes Done() channel is trigger
        cancel()
}

func doStuff(ctx context.Context) {
        if deadline, ok := ctx.Deadline(); ok {
                if time.Now().After(deadline) {
                        return ctx.Err()
                }
        }
        // ... do actual work...
}
```

Value()

- The Value() function provides a way to load request scoped data that has been stored on the context

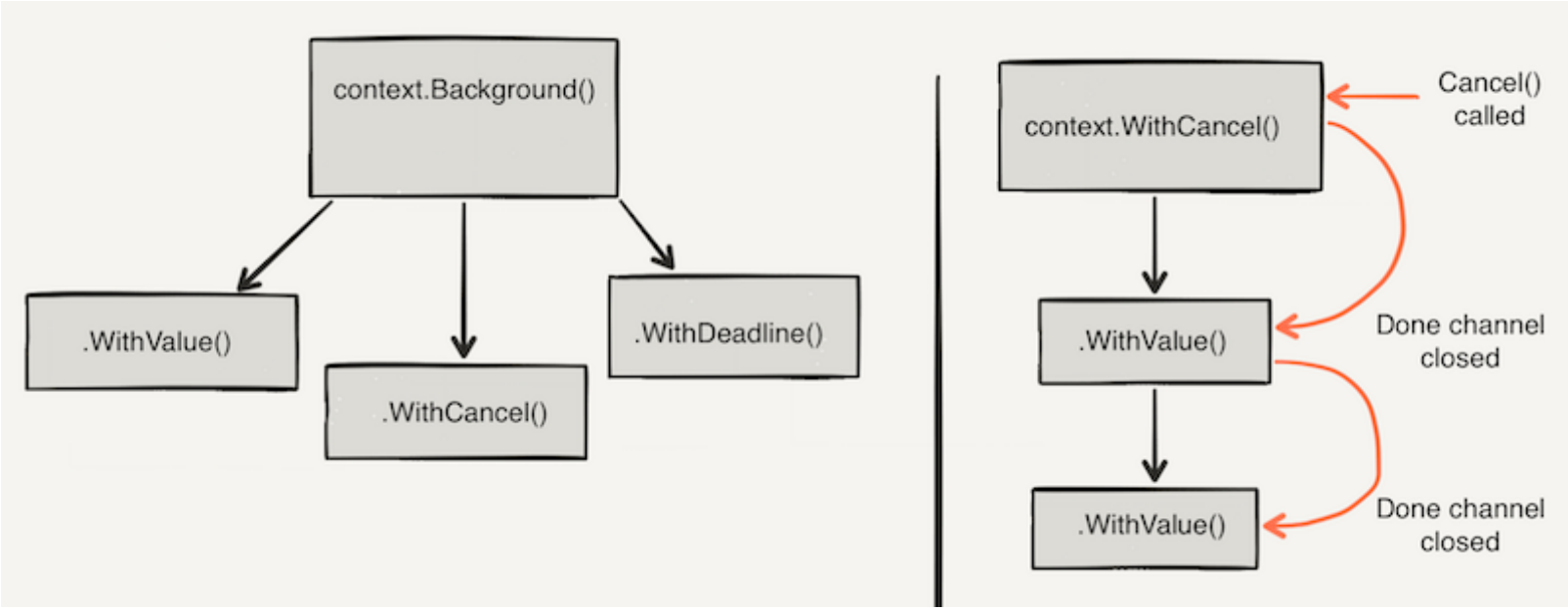
```
var string value = "SomeValue"
ctx = context.WithValue(context.Background(), key, value)
val := ctx.Value(key).(string)
```

A few notes from recent context conversations online:

- Context value handling is completely type unsafe and can't be checked at compile time
- Essentially a `map[interface{}]interface{}`
- Good examples of data to store in context include data extracted from headers or cookies, userID's tied to auth information, etc

Derived Contexts

- The context package provides functions that derive new Context values from existing ones
- These Contexts form a tree and when any Context is cancelled all those dervied from it are also cancelled
- Provides a mechanism to manage the lifecycle of dependent functions within a request scoped operation



Deriving Contexts

Background()

```
func Background() Context
```

- Typically the top level Context for incoming requests

TODO()

```
func TODO() Context
```

- If it's unclear what Context to use or it is not yet available use TODO never send nil for a Context parameter

Deriving Contexts (cont'd)

WithCancel():

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

- Returns a copy of the parent with a new Done channel
- The context's Done channel is closed when the cancel function is called or the parent context Done channel is closed

WithDeadline()

```
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
```

- Takes a time param and returns a copy of the parent context with the deadline adjusted to be no later than the time parameter
- The context's Done channel is closed when the deadline expires, when the returned cancel function is called or when the parent's Done channel is closed (whichever comes first)

Deriving Contexts (cont'd)

WithTimeout()

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

- Returns a context with the deadline set to the current time plus the value of the timeout
- Code should call cancel as soon as operations running this Context complete

WithValue()

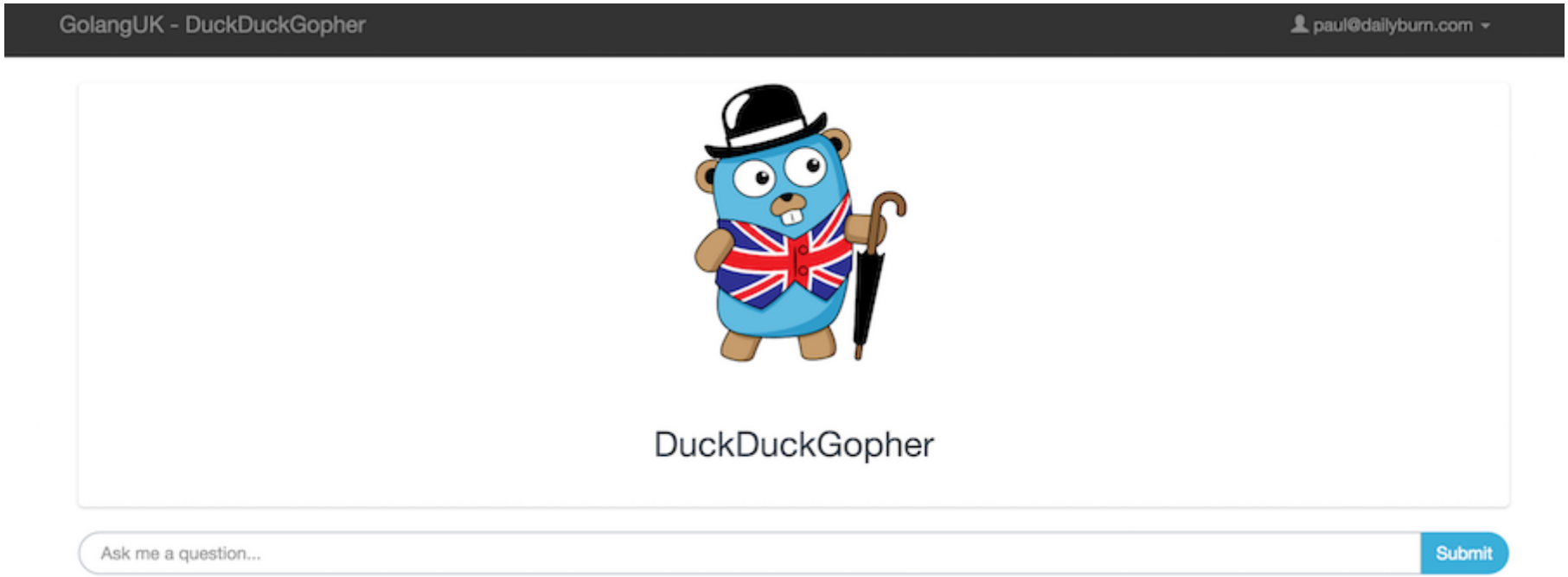
```
func WithValue(parent Context, key interface{}, val interface{}) Context
```

- Returns a copy of the parent in which the value for the specified key is set to val

Demo app

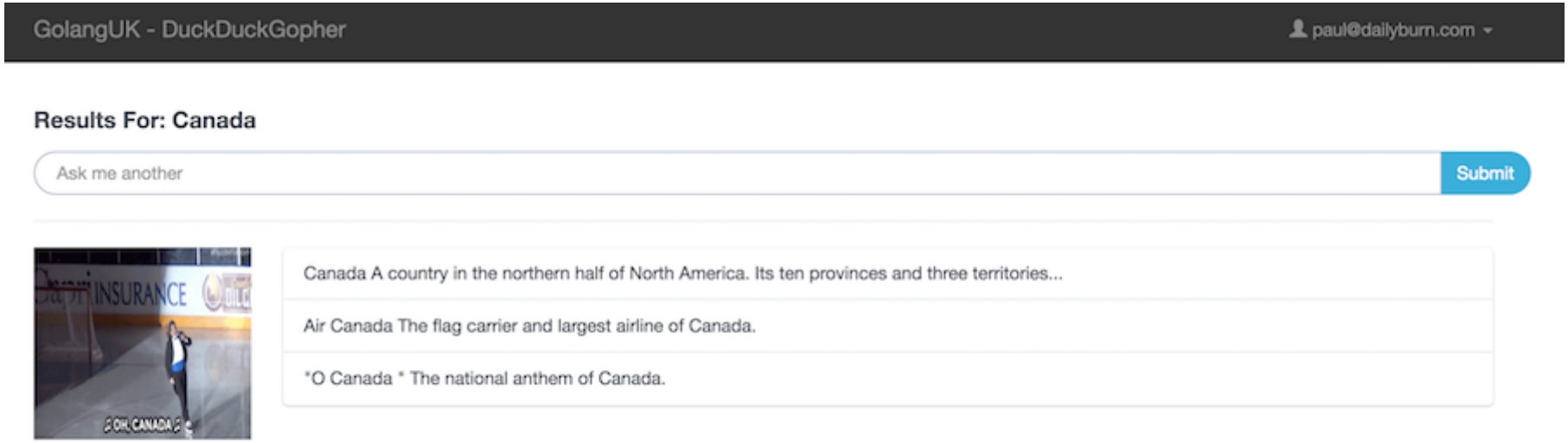
- To demonstrate some of these concepts let's take a look at a small sample app
- We'll build a very simple search engine with a new twist on search

DuckDuckGopher



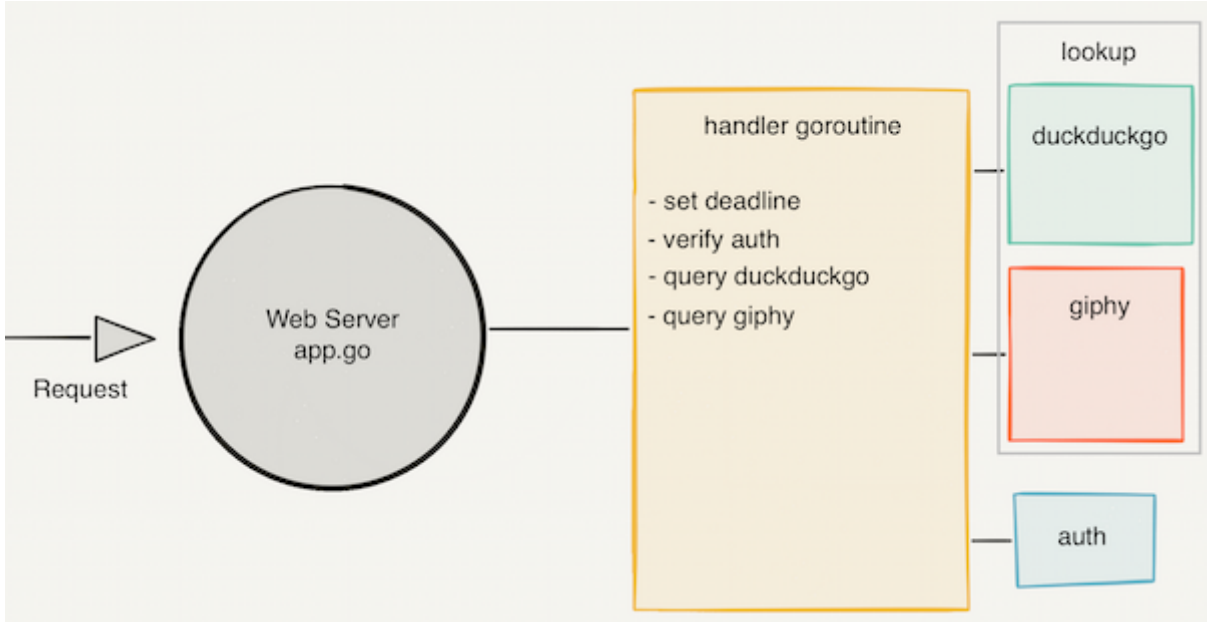
DuckDuckGopher

- So what does our app actually do?
- User types in a search term and gets a page with some results
- User account required for our very exclusive search engine
- Our cutting edge innovation is to provide a gif to visualize the result



So how does it work?

- Search request made to web app
- Authentication cookie checked
- Query call made to DuckDuckGo API
- Keyword search call made to Giphy API
- Results returned to browser



Components

- **Web Server**
- **session** package
- lookup package (DuckDuckGo Answers API, Giphy API)

Web Server

- **app.go** -> HTTP server handles all web requests for the app

Handlers:

authentication:
- **login:** renders login page
- **logout:** destroys active session
- **authenticate:** creates new auth session

search:
- **home:** renders main search page
- **search:** processes search request and returns results

Web Server - authentication

```
func login(w http.ResponseWriter, r *http.Request) {
        templates.ExecuteTemplate(w, "login", nil)
}
```

```
func logout(w http.ResponseWriter, r *http.Request) {
        session.Delete(w, r, store)
        http.Redirect(w, r, "/login", http.StatusFound)
}
```

```
func authenticate(w http.ResponseWriter, r *http.Request) {
        email := r.FormValue("email")
        password := r.FormValue("password")

        if auth.Authenticate(email, password) {
                session.Save(email, w, r, store)
                http.Redirect(w, r, "/", http.StatusFound)
                return
        }

        http.Redirect(w, r, "/login", http.StatusFound)
}
```

session Package (interlude)

```
const userSessionKey string = "user"
const storeKey string = "session-user"

func Save(email string, rw http.ResponseWriter, req *http.Request, store *sessions.CookieStore) error {
	session, err := FromRequest(req, store)
	if err != nil {
		return err
	}

	session.Values[userSessionKey] = email
	return session.Save(req, rw)
}

func Delete(rw http.ResponseWriter, req *http.Request, store *sessions.CookieStore) error {
	session, err := FromRequest(req, store)
	if err != nil {
		return err
	}
	delete(session.Values, userSessionKey)
	return session.Save(req, rw)
}
```

session Package (interlude cont'd)

```
func Email(s *sessions.Session) (string, bool) {
        email, ok := s.Values[userSessionKey].(string)
        if ok {
                return email, true
        }

        return "", false
}

// FromRequest extracts the user email from req, if present.
func FromRequest(req *http.Request, store *sessions.CookieStore) (*sessions.Session, error) {=
        if store == nil {
                return nil, errors.New("Cookie store is nil")
        }

        return store.Get(req, storeKey)
}
```

session Package (interlude cont'd)

```
type key int

const sessionCtxKey key = 0

// NewContext returns a new Context carrying session
func NewContext(ctx context.Context, s *sessions.Session) context.Context {
	return context.WithValue(ctx, sessionCtxKey, s)
}

// FromContext extracts the session from ctx, if present.
func FromContext(ctx context.Context) (*sessions.Session, bool) {
	// ctx.Value returns nil if ctx has no value for the key
	s, ok := ctx.Value(sessionCtxKey).(*sessions.Session)
	return s, ok
}
```

Web Server - home

- home renders the main search page

```
func home(w http.ResponseWriter, r *http.Request) {
        s, err := session.FromRequest(r, store)
        user, ok := session.Email(s)
        if err != nil || !ok {
                http.Redirect(w, r, "/login", http.StatusFound)
                return
        }

        params := map[string]interface{}{
                "user": user,
        }

        templates.ExecuteTemplate(w, "search", params)
}
```

- simple form that posts text entered into search field back to the server

Web Server - search

- the main search handler is where the real meat of our app lives

```
func search(w http.ResponseWriter, r *http.Request) {
        s, err := session.FromRequest(r, store)
        user, ok := session.Email(s)
        if err != nil || !ok {
                http.Redirect(w, r, "/login", http.StatusFound)
                return
        }

        qry := r.FormValue("input")

        // ...
```

- verify authentication and extract the search text entered

Web Server - search cont'd

```
// create a context with a hard deadline for returning something
ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)

type resultAndError struct {
        results []string
        err     error
}

// ask duckduckgo for an answer
answerChan := make(chan resultAndError)
go func() {
        value, err := lookup.DuckduckQuery(ctx, qry)
        answerChan <- resultAndError{value, err}
}()

// ask giphy for a gif
sessionCtx := session.NewContext(ctx, s)
gifChan := make(chan resultAndError)
go func() {
        terms := strings.Split(qry, " ")
        url, err := lookup.GifForTerms(sessionCtx, terms, giphyKey)
        gifChan <- resultAndError{[]string{url}, err}
}()
```

Web Server - search cont'd

```
var results []string
var gif string

func() {
        for {
                select {
                case r := <-answerChan:
                        results = r.results
                        if r.err != nil || len(results) < 1 {
                                results = []string{"Whoops we couldn't find anything!"}
                        }
                        cancel() // We got our main result cancel the context
                        return
                case r := <-gifChan:
                        if r.err != nil {
                                continue
                        }
                        gif = r.results[0]
                case <-ctx.Done():
                        if results == nil {
                                results = []string{"Whoops we ran out of time!"}
                        }
                        return
                }
        }
}()
```

Web Server - search cont'd

```
        params := map[string]interface{}{
                "results":  results,
                "question": qry,
                "gif":      gif,
                "user":     user,
        }

        templates.ExecuteTemplate(w, "results", params)
}
```

```
<div class="col-md-1">
    {{ if .gif }}
        <img src="{{ .gif }}" style="width:90px; height: 90px;">
    {{ else }}
        <img src="/public/img/golang_gopher.svg" style="width:20px;">
    {{ end }}
</div>
<div class="col-md-11">
    <ul class="list-group">
        {{ range .results }}
            <li class="list-group-item">{{ . }}</li>
        {{ end }}
    </ul>
</div>
```

Components

- Web Server
- session package
- **lookup** package (DuckDuckGo Answers API, Giphy API)

Lookup - Duckduckgo

```
func DuckduckQuery(ctx context.Context, question string) ([]string, error) {
        type responseAndError struct {
                resp []string
                err  error
        }

        respChan := make(chan responseAndError)
        go func() {
                resp, err := goduckgo.Query(question)
                if afterDeadline(ctx) {
                        respChan <- responseAndError{nil, ctx.Err()}
                        return
                }

                var result []string
                if resp != nil {
                        result = combineResults(resp)
                }

                respChan <- responseAndError{result, err}
                return
        }()
        // ...
```

Lookup - Duckduckgo (cont'd)

```
        select {
        case r := <-respChan:
                return r.resp, r.err
        case <-ctx.Done(): // if the context is cancelled return
                return nil, ctx.Err()
        }
}

func combineResults(resp *goduckgo.Message) []string {
        // extract and combine data from duckduckgo api
        // return array of result strings
}
```

Lookup - Giphy

```
func GifForTerms(ctx context.Context, terms []string, apiKey string) (string, error) {
	if afterDeadline(ctx) {
		return "", ctx.Err()
	}

	rating := "r" // default rating
	s, ok := session.FromContext(ctx)
	if ok {
		rating = ratingForUser(s)
	}

	termsString := strings.Join(terms, "+")
	params := map[string]interface{}{"api_key": apiKey, "q": termsString, "rating": rating}
	resp, err := getGiphy(ctx, apiPath, params)
	if err != nil {
		return "", err
	}

	url, perr := parseResponse(resp)
	if perr != nil {
		return "", perr
	}

	return url, nil
}
```

Lookup - Giphy (cont'd)

```
func afterDeadline(ctx context.Context) bool {
        if deadline, ok := ctx.Deadline(); ok {
                if time.Now().After(deadline) {
                        return true
                }
        }

        return false
}

func parseResponse(resp *http.Response) (string, error) {
        // parses the json response from giphy to extract a displayable url
}

func ratingForUser(s *sessions.Session) string {
        // returns the giphy rating for the user session
}
```

Lookup - Giphy (cont'd)

```
func getGiphy(ctx context.Context, path string, params map[string]interface{}) (*http.Response, error) {
        var requestUrl string

        if params != nil {
                queryParams := url.Values{}
                for k, v := range params {
                        queryParams.Add(k, v.(string))
                }
                requestUrl = path + "?" + queryParams.Encode()
        } else {
                requestUrl = path
        }

        client := http.Client{}
        return ctxhttp.Get(ctx, &client, requestUrl)
}
```

Lookup - ctxhttp

- **ctxhttp:** we're using **ctxhttp.Get** to make our http call to giphy

```
// under the hood it calls: (pre Go 1.7)
func Do(ctx context.Context, client *http.Client, req *http.Request) (*http.Response, error) {
        if client == nil {
                client = http.DefaultClient
        }

        // Request cancelation changed in Go 1.5, see cancelreq.go and cancelreq_go14.go.
        cancel := canceler(client, req)

        type responseAndError struct {
                resp *http.Response
                err  error
        }
        result := make(chan responseAndError, 1)

        go func() {
                resp, err := client.Do(req)
                result <- responseAndError{resp, err}
        }()
```

Lookup - ctxhttp (cont'd)

```
        select {
        case <-ctx.Done():
                cancel()
                return nil, ctx.Err()
        case r := <-result:
                return r.resp, r.err
        }
}
```

Lookup - ctxhttp (cont'd)

```
// Here's the version from Go 1.7 just for fun
func Do(ctx context.Context, client *http.Client, req *http.Request) (*http.Response, error) {
        if client == nil {
                client = http.DefaultClient
        }
        resp, err := client.Do(req.WithContext(ctx))
        // If we got an error, and the context has been canceled, the context's error is probably more u
        if err != nil {
                select {
                case <-ctx.Done():
                        err = ctx.Err()
                default:
                }
        }
        return resp, err
}
```

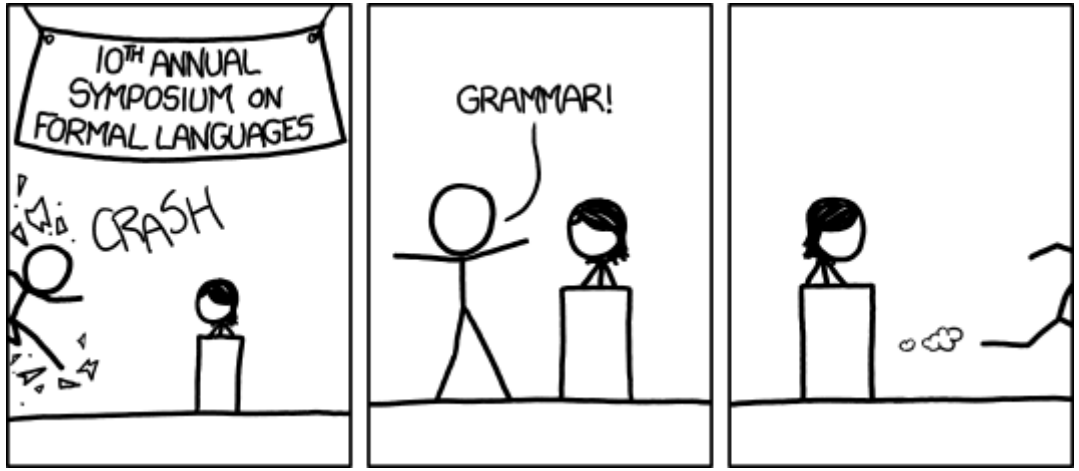
DuckDuckGopher - Demo

- Let's see it in action

Thanks

Paul Crawford

paul@dailyburn.com
@paulcrawford



Thank you