Разработка веб-сервисов на Go, часть 2. Лекция 1

Middleware

Основы middleware для HTTP

В этом разделе мы поговорим про Middleware — промежуточный код, который располагается между запросом пользователя и вашей бизнес-логикой. В общем случае Middleware является любой фреймворк, потому что он осуществляет какой-то набор общих операций, общих проверок. Мы же обсудим, как можно реализовать базовый функционал Middleware не используя ничего кроме стандартной библиотеки. Для начала рассмотрим пример того. как делать не надо.

```
func pageWithAllChecks(w http.ResponseWriter, r *http.Request) {
        defer func() {
                if err := recover(); err != nil {
                        fmt.Println("recovered", err)
                        http.Error(w, "Internal server error", 500)
                }
        }()
        defer func(start time.Time) {
                fmt.Printf("[%s] %s, %s %s\n",
                        r.Method, r.RemoteAddr, r.URL.Path, time.Since(start))
        }(time.Now())

        _, err := r.Cookie("session_id")
        // учебный пример! это не проверка авторизации!
        if err != nil {
                fmt.Println("no auth at", r.URL.Path)
                http.Redirect(w, r, "/", http.StatusFound)
                return
        }

        // your logic
}
```

Перед вами функция pageWithAllChecks, в которой реализованы все нужные проверки. В начале функции мы обрабатываем панику — если вдруг паника была вызвана где-то в глубине, то её обязательно нужно перехватить до завершения функции, чтобы процесс не упал. Дальше идет печать accessLog, мы печатаем, какой тип запроса, url, с какого адреса он пришел, сколько времени занял. AccessLog тоже нужен всегда, чтобы понимать, что происходит во время работы. Дальше идет проверка авторизации, в учебном примере мы просто проверяем наличие cookie для простоты. Весь этот код нужен вне зависимости от того, что в функции будет происходить дальше. Однако, если копипастить его из функции в функцию, то, во-первых, код слишком разрастется из-за однотипных кусков, во-вторых, вносить изменения в такой код, ничего не пропустив, будет довольно сложно.

Теперь рассмотрим, каким образом можно избавиться от дублирования. Создадим админский мультиплексор, который будет обрабатывать страницу адмна и демо-страницу с паникой. На этих страницах с префиксом admin нужна авторизация, поэтому мы завернем мультиплексор в adminAuthMiddleware, на выходе получим Handler. Сделаем корневой мультплексор запросов. Теперь в кажестве обработчика страниц с префиксом admin мы передадим уже handler, который будет идти с авторизацией. Дальше у нас будут страницы login и logout, и в конце мы установим еще какой-то middleware для печати accessLog и печати panic. Окончательную версию siteHandler передадим уже непосредственно в вебсервер, который будет обрабатывать запросы.

```
func main() {
        adminMux := http.NewServeMux()
        adminMux.HandleFunc("/admin/", adminIndex)
        adminMux.HandleFunc("/admin/panic", panicPage)

        // set middleware
        adminHandler := adminAuthMiddleware(adminMux)

        siteMux := http.NewServeMux()
        siteMux.Handle("/admin/", adminHandler)
        siteMux.HandleFunc("/login", loginPage)
        siteMux.HandleFunc("/logout", logoutPage)
        siteMux.HandleFunc("/", mainPage)

        // set middleware
        siteHandler := accessLogMiddleware(siteMux)
        siteHandler = panicMiddleware(siteHandler)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", siteHandler)
}
```

Каким образом это устроено? Начнем мы с panicMiddleware.

```
func panicMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                fmt.Println("panicMiddleware", r.URL.Path)
                defer func() {
                        if err := recover(); err != nil {
                                fmt.Println("recovered", err)
                                http.Error(w, "Internal server error", 500)
                        }
                }()
                next.ServeHTTP(w, r)
        })
}
```

Мы передаем в функцию http.Handler, то есть это может быть любой Handler или даже мультиплексор запросов. Возвращаем тоже http.Handler. И тут происходит вся магия. Мы возвращаем функцию, которая будет принимать входящий запрос в себя, то есть возвращаемая функция теперь будет отрабатывать первой. В ней мы сделаем в defer обработку паники, а потом вызовем следующую функцию. То есть функции будут вызываться цепочкой. Кроме того, мы сделали общую функцию и обернули все обработчики в нее, теперь все обработчики будут проходить через panicMiddleware. Аналогично будут реализованы accessLogMiddleware и adminAuthMiddleware. Возвращается функция-хендлер, в которой сначала выполняется какая-то дополнительная работа, а затем вызывается обработчик, переданный на вход. При каждом запросе будет отрабатывать panicMiddleware, после этого accessLogMiddleware, а после, если мы на админской странице, обработает еще третий запрос, adminAuthMiddleware. То есть мы построили такую цепочку функций, которая вызывается одна за другой и при этом избавились от копипаста. Обратите внимание, panic вызывается последней, потому что выполняться они будут в обратном порядке.

Теперь осталось не забыть реализовать сами функции с бизнес-логикой. Но в них не происходит ничего интересного, поэтому изучите их самостоятельно в коде к лекции.

Попробуем запустить сервер и походить между страничками. Получим какой-то такой лог, с его помощью

легко проследить, что вызовы происходят ровно так, как мы ожиидаем: обработка паники, печать логов, проверка авторизации (если требуется), обработка запроса.

```
starting server at :8080
panicMiddleware /
accessLogMiddleware /
[GET] 127.0.0.1:60268, / 13.489µs
panicMiddleware /login
accessLogMiddleware /login
[GET] 127.0.0.1:60268, /login 63.244µs
panicMiddleware /
accessLogMiddleware /
[GET] 127.0.0.1:60268, / 39.417µs
panicMiddleware /logout
accessLogMiddleware /logout
[GET] 127.0.0.1:60268, /logout 226.496µs
panicMiddleware /
accessLogMiddleware /
[GET] 127.0.0.1:60268, / 12.125µs
panicMiddleware /admin
accessLogMiddleware /admin
[GET] 127.0.0.1:60268, /admin 18.402µs
panicMiddleware /admin/
accessLogMiddleware /admin/
adminAuthMiddleware /admin/
no auth at /admin/
[GET] 127.0.0.1:60268, /admin/ 27.464µs
panicMiddleware /
accessLogMiddleware /
[GET] 127.0.0.1:60268, / 13.991µs
panicMiddleware /login
accessLogMiddleware /login
[GET] 127.0.0.1:60268, /login 40.882µs
panicMiddleware /
accessLogMiddleware /
[GET] 127.0.0.1:60268, / 18.873µs
panicMiddleware /admin/
accessLogMiddleware /admin/
adminAuthMiddleware /admin/
[GET] 127.0.0.1:60268, /admin/ 25.227µs
panicMiddleware /
accessLogMiddleware /
[GET] 127.0.0.1:60268, / 18.822µs
panicMiddleware /admin/panic
accessLogMiddleware /admin/panic
```

Context value

Продолжим говорить про middleware и рассмотрим, каким образом через middleware можно организовать тайминги каких-либо операций, происходящих внутри запроса. Для этого мы обратимся к уже известному вам пакету «контекст», ранее мы его рассматривали только в аспекте отмены запросов, однако это не единственная его возможность. Кроме этого есть возможность нести в себе какие-то значения, то есть context value. Причем контекст будет копироваться при перезаписи. То есть это что-то вроде tread safe local storage, но для конкретной области видимости — для вашего запроса.

Рассмотрим в коде, как с этим работать. Рассмотрим функцию main, тут ничего сложного нет, мы делаем multiplex запросов, регистрируем через обработчик, регистрируем там один middleware и начинаем запрос.

```
func main() {
        rand.Seed(time.Now().UTC().UnixNano())

        siteMux := http.NewServeMux()
        siteMux.HandleFunc("/", loadPostsHandle)

        siteHandler := timingMiddleware(siteMux)

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", siteHandler)
}
```

Вроде ничего сложного. Но прежде чем рассматривать, как устроен внутри timingMiddleware, посмотрим, что происходит в хендлере loadPostsHandle.

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

        emulateWork(ctx, "checkCache")
        emulateWork(ctx, "loadPosts")
        emulateWork(ctx, "loadPosts")
        emulateWork(ctx, "loadPosts")
        time.Sleep(10 * time.Millisecond)
        emulateWork(ctx, "loadSidebar")
        emulateWork(ctx, "loadComments")

        fmt.Fprintln(w, "Request done")
}

func emulateWork(ctx context.Context, workName string) {
        defer trackContextTimings(ctx, workName, time.Now())

        rnd := time.Duration(rand.Intn(AvgSleep))
        time.Sleep(time.Millisecond * rnd)
}
```

Итак, сначала мы получаем контекст из нашего запроса, затем выполняем несколько раз небольшую функцию, которая будет эмулировать работу. В нее каждый раз передаем контекст первым параметром — контекст всегда должен передаваться первым параметром, это такое соглашение — и какое-то имя работы. И еще делаем небольшой Sleep без эмуляции работы и выводим, что запрос выполнен. Внутри emulateWork тоже все очень просто. Через defer вызываем функцию trackContextTimings, куда передаем контекст, имя работы и время начала запроса. Стоит вспомнить, что в defer аргумент вычисляется в момент объявления defer, то есть это время начала действительно вычислится до начала работы , а функция выполнится после sleep.

Теперь посмотрим, как все реализовано внутри. Начнем мы с timingMiddleware.

```
func timingMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                ctx := r.Context()
                ctx = context.WithValue(,
                        timingsKey,
                        &ctxTimings{
                                Data: make(map[string]*Timing),
                        })
                defer logContextTimings(ctx, r.URL.Path, time.Now())
                next.ServeHTTP(w, r.WithContext(ctx))
        })
}
```

Сначала точно так же получаем контекст из запроса. Затем добавляем еще значение в него, при помощи функции withValue, в которой указываем ключ, по которому будем к этому значению обращаться, и свою структуру, в которой будем значение хранить. У нас создался новый индекс контекста, в котором появилось новое значение, при этом старый контекст не изменился, потому что copy on right. Затем в defer вызываем logContextTimings, куда опять передаем этот самый контекст, url и время начала запроса. А дальше продолжаем обслуживать наш запрос. То есть, когда начнет выполняться сам обработчик запроса в нем уже будет контекст со временем начала работы внутри.

Вроде бы должна уже вырисовываться картина, как это работает. Осталось только разобрать функцию trackContextTimings.

```
type Timing struct {
        Count    int
        Duration time.Duration
}
```

```
type ctxTimings struct {
        sync.Mutex
        Data map[string]*Timing
}

func trackContextTimings(ctx context.Context, metricName string, start time.Time) {
        // получаем тайминги из контекста
        // поскольку там пустой интерфейс, то нам надо преобразовать к нужному типу
        timings, ok := ctx.Value(timingsKey).(*ctxTimings)
        if !ok {
                return
        }
        elapsed := time.Since(start)
        // лочимся на случай конкурентной записи в мапку
        timings.Lock()
        defer timings.Unlock()
        // если меткри ещё нет - мы её создадим, если есть - допишем в существующую
        if metric, metricExist := timings.Data[metricName]; !metricExist {
                timings.Data[metricName] = &Timing{
                        Count:    1,
                        Duration: elapsed,
                }
        } else {
                metric.Count++
                metric.Duration += elapsed
        }
}
```

В ней мы получаем тайминг из контекста, используя ключ, и как-то его обрабатываем. Стоит обратить внимание на то, что контекст — хранилище такое универсальное, что хранится там все в виде пустого интерфейса. Поэтому, получив value его сразу же нужно преобразовывать к нужному типу данных. Так мы и поступаем — проверяем, удается ли преобразовать. Если удается, смотрим, сколько прошло времени, лочимся, внутри структуры мапка — не горутинобезопасный тип данных, в defer вызываем unlock и дальше проверяем, если текущая метрика уже есть,то плюсуем время и количество, если нет, добавляем ее в map.

Мы посмотрели, каким образом мы добавляем в тайминг. То есть мы получили нашу структуру заплюсовали туда и все. Теперь вернемся к функции Middleware, вспомним, что в defer вызываем логирование. Рассмотрим это логирование.

```
func logContextTimings(ctx context.Context, path string, start time.Time) {
        // получаем тайминги из контекста
        // поскольку там пустой интерфейс, то нам надо преобразовать к нужному типу
        timings, ok := ctx.Value(timingsKey).(*ctxTimings)
        if !ok {
                return
        }
        totalReal := time.Since(start)
        buf := bytes.NewBufferString(path)
        var total time.Duration
        for timing, value := range timings.Data {
                total += value.Duration
                buf.WriteString(fmt.Sprintf("\n\t%s(%d): %s", timing, value.Count, value.Durati
        }
        buf.WriteString(fmt.Sprintf("\n\ttotal: %s", totalReal))
        buf.WriteString(fmt.Sprintf("\n\ttracked: %s", total))
        buf.WriteString(fmt.Sprintf("\n\tunkn: %s", totalReal-total))

        fmt.Println(buf.String())
}
```

Опять-таки получаем тайминги из контекста и проверяем, что они там есть и они действительно скастовались к тому типу данных, который нам нужен. Теперь смотрим время начала запроса, и итерируемся по всем таймингам. Плюсуем каждый учтенный промежуток времени к duration и выводим, к какой работе относится этот тайминг, сколько таких работ было и сколько времени они занимали. Напоследок, считаем, сколько всего времени занял наш запрос и вычисляем неучтенное время, чтобы не упустить, например, тот самый Sleep без эмуяции работы.

Еще раз я вам напомню: Context Value предназначен для того, чтобы хранить в request scope какое-то значение на протяжении жизни вашего запроса. В нем не надо хранить то, что не уничтожается после того, как запрос отработает. То есть хранить подключение к базе данных, например, там не надо. Также через Context Value нельзя ни в коем случае передавать параметры в функцию, потому что Context Value работает через пустой интерфейс, значит, он создает недокументированное runtime api, которое нельзя проверить в compile time. То есть если у вас есть четкая структура, вы знаете, что куда можно присвоить, то есть четкие проверки. В Context Value проверок нет, все нужно проверять руками. Также контекст никогда не надо присваивать уже в какую-то готовую структуру, разве что это ваш отдельный контекст, в который вы его присваиваете через композицию. То есть контекст всегда просто контекст, он не часть структуры, он всегда передается как первый аргумент в функцию, контекст не должен быть параллельным API. Контекст очень полезен, но с ним нужно быть очень осторожным и хранить там только те значения, которые нужны вам в течение всего срока жизни вашего запроса. Например, тайминги или сессию или requestID или userID.

Обработка ошибок

В предыдущих учебных примерах мы часто возвращали ошибки из функций, иногда просто выводили их на экран, иногда как-то обрабатывали, иногда просто игнорировали. Настало время поговорить подробнее о том, что такое ошибка, как лучше ее писать, и как лучше ее обрабатывать.

Представим, что у нас есть веб-сервис, которых ходит на какой-то удаленный ресурс по http и что-то парсит. В данном случае неважно что, результат мы тут использовать не будем, нам главное ошибка.

```
var (
        client = http.Client{Timeout: time.Duration(time.Millisecond)}
)

func getRemoteResource() error {
        url := "http://127.0.0.1:9999/pages?id=123"
        _, err := client.Get(url)
        if err != nil {
                // вернётся `timed out`. и что?
                // return err

                // будет `res error: time out`. а где?
                // return fmt.Errof("getRemoteResource: %+v", err)

                return fmt.Errorf("getRemoteResource: %s at %s", err, url)
        }
        return nil
}
```

Создадим клиент http.Client (используем не стандартный клинт, чтоб можно было установить тайм-аут), в который устанавим очень маленький тайм-аут — одну миллисекунду. Плюс будем ходить на сервис, где никто не слушает порт. Вызываем у клиента метод get, передаем туда url какой-то, который, допустим, меняется внутри нашего приложения периодически. Проверяем, вернулась ли ошибка. Раньше в случае ошибки мы сразу завершали функцию — писали return err. Так поступать бывает довольно плохо, потому что по такой, ни во что не обернутой ошибке в логах, например, будет почти невозможно восстановит где она произошла, понять какой, например, ресурс зафейлился или какое из множества соединений отвалилось. Чаще ошибку возвращают не сразу, а оборачивают её еще в одну, уже с каким-то префиксом. В данном случае можно написать название функции, откуда вернется ошибка, а затем саму ошибку. Причем ее можно интерпретировать как строку, а можно выводить полностью как тип данных. Так же было бы удобно видеть не только текст ошибки, но еще параметры, с которыми она произошла. Например, в данном случае выводим еще и url, чтобы можно было определить не только, что ошибка произошла в этой функции, но еще и узнать, какой именно ресурс сфейлился.

Мы рассмотрели, что можно делать, чтоб ошибку было проще локализовать. Как поступать, если мы хотим выяснить, какая ошибка вернулась. Например, хотим точно знать, нам вернулся таймаут или что-то еще. То есть мы хотим иметь своего рода типизированную ошибку. В пакете errors есть функция New, с помощью которой можно создать новую ошибку.

```
var (
        client = http.Client{Timeout: time.Duration(time.Millisecond)}

        ErrResource = errors.New("resource error")
)

func getRemoteResource() error {
        url := "http://127.0.0.1:9999/pages?id=123"
        _, err := client.Get(url)
        if err != nil {
                return ErrResource
        }
        return nil
}
```

Именно эту переменную будем возвращать, если поймали ошибку на ресурсе. а на другой стороне сможем эту переменную проверить.

```
func handler(w http.ResponseWriter, r *http.Request) {
        err := getRemoteResource()
        if err != nil {
                fmt.Printf("error happend: %+v\n", err)
                switch err {
                case ErrResource:
                        http.Error(w, "remote resource error", 500)
                default:
                        http.Error(w, "internal error", 500)
                }
                return
        }
        w.Write([]byte("all is OK"))
}
```

Таким образом определить, какого типа была ошибка. И уже в зависимости от этого написать internal error, либо сказать, что это не просто какая-то непонятная ошибка, а ошибка доступа к удаленному ресурсу. То есть ошибка у меня уже принимает какой-то тип. Однако в этом случае теряется контекст. Я уже не могу понять, на каком url она произошло.

Пойдем дальше. Самое время разобраться, что такое ошибка. Error на самом деле это интерфейс, у которого должна быть функция Error, которая возвращает строчку. Любая ваша структура, которая подойдет под этот интерфейс, может возвращаться в качестве ошибки. Напишем свою типизированную ошибку ResourceError.

```
type ResourceError struct {
        URL string
        Err error
}

func (re *ResourceError) Error() string {
        return fmt.Sprintf(
                "Resource error: URL: %s, err: %v",
                re.URL,
                re.Err,
        )
}
```

У нее будет уже URL как поле структуры, поле под ошибку, которая там в реальности произошла и функция Error, которая и делает нашу структуру ошибкой, выводит текст. Теперь, когда у нас произойдет ошибка на ресурсе, мы вернем типизированную ошибку.

```
func getRemoteResource() error {
        url := "http://127.0.0.1:9999/pages?id=123"
        _, err := client.Get(url)
        if err != nil {
                return &ResourceError{URL: url, Err: err}
        }
        return nil
}
```

Это уже будет не просто именованная какая-то переменная, а прямо уже целая структура, которая содержит в себе информацию о том, на каком месте произошла эта ошибка, и сам текст ошибки. Кроме того дальше мы сможем различать наши ошибки уже по типам. В этом нам поможет type casting, то есть мы будем пытаться привести к чему-то интерфейс и определить, какого он типа. Это делается с помощью конструкции switch err.(type).

```
func handler(w http.ResponseWriter, r *http.Request) {
        err := getRemoteResource()
        if err != nil {
                switch err.(type) {
                case *ResourceError:
                        err := err.(*ResourceError)
                        fmt.Printf("resource %s err: %s\n", err.URL, err.Err)
                        http.Error(w, "remote resource error", 500)
                default:
                        fmt.Printf("internal error: %+v\n", err)
                        http.Error(w, "internal error", 500)
                }
                return
        }
        w.Write([]byte("all is OK"))
}
```

Теперь, если мы попадаем в первую ветку, то можем привести интерфейс к определенному типу и получить доступ еще и к полям структуры. В случае, если вернулась любая другая ошибка, я выведем, что там случилась internal error — какая-то неизвестная ошибка.

Иногда бывают ситуации, когда хочется выводить стек трейс и корректно оборачивать ошибку в ошибку. Для этих целей есть пакет pkg/errors на GitHub. Он не встроен в Go, это внешний пакет, его нужно устанавливать через Go get. В нем есть возможность обернуть ошибку в какой-то текст.

```
package main

import (
        "fmt"
        "net/http"
        "net/url"
        "time"

        "github.com/pkg/errors"
)


func getRemoteResource() error {
        url := "http://127.0.0.1:9999/pages?id=123"
        _, err := client.Get(url)
        if err != nil {
                return errors.Wrap(err, "resource error")
        }
        return nil
}
```

Таким образом, можно не типизировать ошибки, не сохранять ошибки для каждого конкретного случая и иметь какой-то контекст для того, чтобы определить, что там внутри произошло, какого типа там была

вообще ошибка. В отличае от fmt.Errorf получается не строка, которую пришлось бы разбирать, а новая ошибка, но уже с информацией-строкой, о том, где она произошла. Кроме того сам тип изначальной ошиби сохраняется. На самом деле http.Client возвращает urlError в случае таймаута. И мы так же с помощью case можем достучаться до внутренней ошибки и проверить, что произошло.

```
func handler(w http.ResponseWriter, r *http.Request) {
        err := getRemoteResource()
        if err != nil {
                fmt.Printf("full err: %+v\n", err)
                switch err := errors.Cause(err).(type) {
                case *url.Error:
                        fmt.Printf("resource %s err: %+v\n", err.URL, err.Err)
                        http.Error(w, "remote resource error", 500)
                default:
                        fmt.Printf("%+v\n", err)
                        http.Error(w, "parsing error", 500)
                }
                return
        }
        w.Write([]byte("all is OK"))
}
```

Раньше мы оборачивали ошибку во что-то свое, а теперь у нас есть возможность делать это из готового пакета. Более того в этом пакете есть возможность вывести полную ошибку, то есть в пакете есть стек трейс. pkg/errors — это довольно распространенный способ работы с ошибками, он используется очень многими пакетами в Go, и он рекомендуется к использованию.

Роутеры

Роутеры - gorilla/mux, httprouter

В этой главе мы поговорим про альтернативные роутеры. Дело в том, что стандартный мультиплексор, несмотря на то, что весьма производителен, довольно ограничен в своих функциях. Например, в нем нельзя указать, для какого метода должен работать заданный url, также с его помощью нельзя парсить параметры напрямую из url'а. Однако иногда это бывает очень нужно, если мы хотим сделать — человекопонятный url, либо же у нас web-приложение построено по архитектуре REST. В этом случае, стоит обратиться к альтернативным реализациям.

Первая из рассматриваемых — это gorilla/mux. Gorilla — это набор web-компонентов, помимо роутера в нем есть еще довольно много всего интересного. У роутера есть возможность делать ограничения по методам, по хостам, по даже query-параметрам, либо можно написать какую-нибудь свою функцию для того, чтобы определять, куда пускать этот роутер или нет. Посмотрим, как это выглядит в коде.

```
package main

import (
        "fmt"
        "log"
        "net/http"

        "github.com/gorilla/mux"
)

func main() {
        r := mux.NewRouter()
        r.HandleFunc("/", List)
        r.HandleFunc("/users", List).
                Host("localhost")
        r.HandleFunc("/users", Create).
                Methods("PUT")
        r.HandleFunc("/users/{id:[0-9]+}", Get)
        r.HandleFunc("/users/{login}", Update).
```

```
                Methods("POST").
                Headers("X-Auth", "test")

        fmt.Println("starting server at :8080")
        log.Fatal(http.ListenAndServe(":8080", r))
}

func Get(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        fmt.Fprintf(w, "you try to see user %s\n", vars["id"])
}

...
```

Для начала нужно импортировать, собственно, gorilla/mux. Это внешний пакет, а не стандартная библиотека. А дальше все почти то же самое. Создаем новый роутер. У него есть метод HandleFunc, в который можно передавать свою функцию-обработчик, указывать, для какого хоста она будет работать, для каких методов, либо даже для каких header'ов, можно указывать параметры, которые мы хотим распарсить из url'а. Для того чтобы получить значение, я должен обратиться к функции получения значений из request.

```
vars := mux.Vars(r)
```

Этот роутер полностью взаимозаменяемый со стандартным в том плане, что параметры, которым они передают запросы, полностью совместимы. Однако, если посмотреть на benchmark'и роутеров, то Gorilla Mux один из самых медленных. У него очень много возможностей, однако наличие этих возможностей имеет свою цену.

Иногда нам нужны возможности, а иногда нам нужно не очень много возможностй и хорошая скорость. Давайте рассмотрим еще один роутер, HttpRouter. Он очень производительный. Он построен на технологии Prefix Tree, или Radix Tree. Итак, смотрим код.

```
package main

import (
        "fmt"
        "log"
        "net/http"

        "github.com/julienschmidt/httprouter"
)

func main() {
        router := httprouter.New()
        router.GET("/", List)
        router.GET("/users", List)
        router.PUT("/users", Create)
        router.GET("/users/:id", Get)
        router.POST("/users/:login", Update)

        fmt.Println("starting server at :8080")
        log.Fatal(http.ListenAndServe(":8080", router))
}

func Get(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
        fmt.Fprintf(w, "you try to see user %s\n", ps.ByName("id"))
}

...
```

После создания роутера, можно прямо указать параметр, на котором будет работать тот или иной метод, можно указать, что мы хотим распарсить какую-то часть url'а в переменную. Однако, в отличае от гориллы, здесь нет матчинга по регулярке. Кроме того у этого роутера другая сигнатура для handler'ов.

Добавляется еще четвертое значение Params. Это, конечно, можно обойти, сделав обертку над нестандартными методами, или прослойку с промежуточным кодом. Самое же главное в этом роутере то, что он очень быстро работает.

Как быть, когда не хочется выбирать между возможностями и скоростью, а нужно реализовать в одном месте скорость, а в другом возможности? Вы не ограничены в выборе только одного роутера, вы можете испоьзовать их столько, сколько вам надо. Вот пример, который совмещает сразу все три роутера. Быстрый роутер, который будет обрабатывать только url /fast/ (допустим, туда идет большая нагрузка, и вам нужно парсить параметры из url'a), следующий роутер, на который будет поступать не очень много запросов, но он должен реализовывать какую-то сложную логику, например фильтровать по хосту, или по заголовкам или, как в примере, разрешать только ajax-запросы. И, наконец, стандартный роутер, для ситуаций, когда не нужно парсить запросы, но нужно работать очень быстро.

```
func main() {
        fastApiHandler := httprouter.New()
        fastApiHandler.GET("/fast/:id", FastRequest)

        complexApiHandler := mux.NewRouter()
        complexApiHandler.HandleFunc("/complex/", ComplexRequest).
                Headers("X-Requested-With", "XMLHttpRequest") // ajax

        stdApiHandler := http.NewServeMux()
        stdApiHandler.HandleFunc("/std/", RegularRequest)

        siteMux := http.NewServeMux()
        siteMux.Handle("/fast/", fastApiHandler)
        siteMux.Handle("/complex/", complexApiHandler)
        siteMux.Handle("/std/", stdApiHandler)

        fmt.Println("starting server at :8080")
        log.Fatal(http.ListenAndServe(":8080", siteMux))
}
```

Эти три роутера можно объединить воедино, используя стандартный мультиплексор. Теперь разные части url'ов будут обрабатываться разными роутерами, за счет этого в одних местах будет получена производительность, а в других простота реализации сложной логики.

Производительный веб-сервер fasthttp

Раз мы заговорили про оптимизацию роутеров, обсудим, что делать, если вас не устроит производительность стандартного http-сервера, который поставляется из коробки. Мы рассмотрим альтернативную реализацию веб-сервера — fasthttp. Он гораздо быстрее и тратит меньше памяти на запрос за счет активного переиспользования всех структур веб-сервера: данных ответа, контекста и прочего.

Поскольку это отдельный абсолютно веб-сервер, он не предоставляет своего мультиплексора, однако уже есть под него порты других. Например, fasthttprouter — это порт роутера httprouter, который мы рассматривали ранее, под fasthttp. Также у fasthttp полностью отличается сигнатура функций, полностью отличается работа с данными запроса. Например, у него нет привычного нам реквеста, в функцию передается только контекст, в котором содержится уже все нужное.

```
package main

import (
        "encoding/json"
        "fmt"
        "log"

        "github.com/buaazp/fasthttprouter"
        "github.com/valyala/fasthttp"
)

func Index(ctx *fasthttp.RequestCtx) {
```

```
        ctx.SetContentType("application/json")
        ctx.SetStatusCode(fasthttp.StatusOK)

        users := []string{"rvasily"}
        body, _ := json.Marshal(users)

        ctx.SetBody(body)
}

func GetUser(ctx *fasthttp.RequestCtx) {
        fmt.Fprintf(ctx, "you try to see user %s\n", ctx.UserValue("id"))
}

func main() {
        router := fasthttprouter.New()
        router.GET("/", Index)
        router.GET("/users/:id", GetUser)

        fmt.Println("starting server at :8080")
        log.Fatal(fasthttp.ListenAndServe(":8080", router.Handler))
}
```

Также, что стоит отметить, что при использовании fasthttp-сервера крайне не безопасно после завершения работы обрабатывать что-то в отдельной горутине, передавая в неё контекст, потому что контекст после завершения функции возвращается в пул и будет переиспользован другим запросом. Но, в целом, это очень быстрый сервер и он часто используется. Под него есть свои фреймворки. Однако вам стоит дважды подумать, прежде чем использовать его.

Валидация

Парсинг параметров в структуру и валидация

В этой главе мы рассмотрим, каким образом можно сделать валидацию параметров в Go. Начнём с того, что в Go нет встроенного валидатора. Это связано с тем, что валидация параметров — это очень специфичная вещь и разнится от проекта к проекту, и нет какой-то общепризнанной спецификации. Однако на просторах Интернета очень много разных пакетов для организации подобного рода действа. В данном примере мы рассмотрим два пакета. Валидатор github.com/asaskevich/govalidator, который умеет только валидировать значения, но не может ниоткуда их распарсить. И парсер параметров из url, из urlvalues в структуру github.com/gorilla/schema.

Рассмотрим пример того, что можно валидировать.

```
type SendMessage struct {
        Id        int    `valid:",optional"`
        Priority  string `valid:"in(low|normal|high)"`
        Recipient string `schema:"to" valid:"email"`
        Subject   string `valid:"msgSubject"`
        Inner     string `schema:"-" valid:"-"`
        flag      int
}
```

У нас есть структура SendMessage (представим, что мы какой-то почтовый сервис, и хотим отправить письмо). В struct tags по метке valid указаны поля для валидатора, по метке schema указаны поля для парсинга параметров в эту структуру. В структуре есть поле Id, в нем указано, что оно опционально, то есть оно может быть, либо его может не быть. Поле Priority — это строковое поле, которое может иметь три значения: low, high, normal. Поле Recipient, которое из параметров в url должено прийти из поля to. За это будет отвечать парсер. Валидатор жн должен знать, что в поле должен лежать валидный email. Поле Subject использует наш собственный валидатор, который мы зарегистрируем специально для этих целей. В поле Inner, обратите внимание — оно публичное, мы не хотим ничего парсить и никак его валидировать, поэтому я и в schema и в поле valid указываем минус. Ещё есть поле flag, но для него вообще

ничего не указываем, потому что это приватное поле, а поскольку и валидация, и парсинг, работают через Reflect, то они не смогут достучаться до этого поля, поэтому и указывать какие-то тэги бессмысленно.

Рассмотрим в коде, каким образом все организовано.

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

        w.Write([]byte("request " + r.URL.String() + "\n\n"))

        msg := &SendMessage{}

        decoder := schema.NewDecoder()
        decoder.IgnoreUnknownKeys(true)
        err := decoder.Decode(msg, r.URL.Query())
        if err != nil {
                fmt.Println(err)
                http.Error(w, "internal", 500)
                return
        }
        w.Write([]byte(fmt.Sprintf("Msg: %#v\n\n", msg)))

        _, err = govalidator.ValidateStruct(msg)

        if err != nil {

                if allErrs, ok := err.(govalidator.Errors); ok {
                        for _, fld := range allErrs.Errors() {
                                data := []byte(fmt.Sprintf("field: %#v\n\n", fld))
                                w.Write(data)
                        }
                }

                w.Write([]byte(fmt.Sprintf("error: %s\n\n", err)))
        } else {
                w.Write([]byte(fmt.Sprintf("msg is correct\n\n")))
        }
}
```

Итак, для нагладности в начале выводим саму строчку запроса, чтобы её было видно на экране. Потом создаем сообщение, декодер для того, чтобы распарсить значение из входящих параметров в структуру. Говорим этому декодеру, чтоб он не ругался а просто игнорировал неизвестные поля. Затем собственно декодируем url в сообщение и выводим его, если не произошла ошибка. Затем вызываем ValidateStruct, который возвращает булевую переменную «да» или «нет» и ошибку. Если случилась ошибка, мы пытаемся достучаться при помощи преобразования интерфейса (err — это интерфейс) до реальных ошибок, которые там есть, преобразованием интерфейса. Если преобразование получилось, то итерируемся по ошибкам и выводим полностью эту ошибку на экран. На самом деле мы можем в этом месте вывести просто строку, но это будет довольно неинформативно, а чтобы понять, в каком месте произошла ошибка, нам придется парсить строку руками. Если же нам удалось преобразовать интерфейс, то мы можем достучаться до произвольного текста ошибки и посмотреть, что там была за ошибка.

Не всегда ваши типы данных, даже если и будут иметь стандартный тип, будут влезать в те методы, которые предоставляет стандартный валидатор. Например, через него нельзя проверить динамический enum. Выше мы валидировали статичный список, но с динамикой так уже не получится. Поэтому напоследок покажем, как зарегистрировать свой собственный обработчик. В примере мы регистрируем валидатор "msgSubject". Это происходит в функции init.

```
func init() {
        govalidator.CustomTypeTagMap.Set("msgSubject",
           govalidator.CustomTypeValidator(func(i interface{}, o interface{}) bool {
                subject, ok := i.(string)
                if !ok {
                        return false
```

```
                }
                if len(subject) == 0 || len(subject) > 10 {
                        return false
                }
                return true
        }))
}
```

Делаем мы это с помощью специальной функции из библиотеки, в которую передаем имя валидатора и сам валидатор. В примере валидатор устроен таким образом, что мы получаем в него Subject, пытаемся преобразовать его к строке и проверяем, что длина больше 0 и меньше 10, в противном случае ругаемся.

Попробуйте запустить и посмотреть, как распарсится такой url.

```
http://127.0.0.1:8080/?to=v.romanov@corp.mail.ru&priority=low&
subject=Hello!&inner=ignored&id=12&flag=23
```

Проверьте, что все работает, как мы и описали, выставляя валидные и невалидные параметры.

Как это работает внутри? Работает внутри это через рефлексию, как и вся динамика, практически, в Go. Поэтому есть хоть маленький, но, тем не менее, overhead. Если же вы хотите там делать совсем-совсем быстро, то добро пожаловать в мир кодогенерации, как всегда. Использовать ли стандартный валидатор либо заморачиваться и писать свой, решать вам. Однако лучше всё-таки начать с тула стандартного, и если вдруг вы заметите, что ваша программа тормозит именно из-за него, то только тогда стоит что-то оптимизировать, потому что, как вы знаете, предварительная оптимизация — это зло.

Фреймворки

Фреймворк Beego

В этом разделе мы обзорно поговорим про фреймворк Beego. Beego — один из самых крупных фреймворков в Go, у него огромное community. и очень много плотно интегрированных между собой компонентов, у него есть свое кеширование, логирование, авторизация, УРМ-система, работа с сессиями, то есть это один из самых крупных фреймворков, что вы можете найти в Go.

В комплекте с Beego есть удобная утилита bee, которая позволяет создать скелет проекта всего одной командой. Чтобы сгенерировать код нужно указать команду bee, команду new и папку, в которую нужно сохранить сгенерированный код. С помощью команды bee run можно тут же запустить сгенерированный сервер. Утилита не только запустит сервер, но начнет мониторить файлы на предмет изменения, чтобы сразу же этот сервер перезагрузить.

Сгенирируйте сервер и изучите полученное. В папке mait все очень аскетично, только подключение роутеров и запуск фреймворка. Порт, на котором слушать сервер, сразу же определяет из конфига. Конфиг, стандартные шаблоны и папка со статичными файлами тоже генерируются рамках этого скелета. Есть определение роутеров, в которых указано что для всего корня будем слушать MainController. Для MainController определена функция ServeHttp, которая уже будет переключать запрос, куда нужно. MainController включает в себя структуру Beego контроллера, в которой как раз и определен ServeHttp. Для MainController можно определять функции для основных rest-запросов, get, post, put, delete и так далее. Например, попробуйте сами доопределить функцию Post. И сразу же post-запрос будет вызван и попадет уже именно в эту функцию.

Так же можно создать свой собственный handler. Но этот запрос не будет доступен сам по себе сразу же. Его еще нужно определить в роутер. Роутер по умолчанию будет проксировать в мой контроллер только rest-запросы. Чтобы привязать новый роутер к урлу, например, custom, нужно добавитть строчку

```
beego.Router("/custom", &controllers.MainController{}, "*:Custom")
```

Последний аргумент означает, что туда будут идти все методы (звездочка),и Custom. В beego есть сложные, в том числе автоматические роутеры, когда вы можете указать только имя вашего контроллера, который вы хотите использовать и через рефлексию все подтянется само.

С помощью bee можно сгененрировать не только html-сайт, но и api-сайт. Для этого нужно запустить

команду bee с командой api и именем папки, куда нужно сгенерировать код. Запускать нужно тоже несколько с другими параметрам.

```
bee run -downdoc=true -gendoc=true
```

Базово ничего особенно интересного нет, только регистрация контроллеров в name spaces, по которым будут сгенерированы автоматически все методы. То есть используется кодогенерация в том числе. Но интересно обратить внимание на мета-информацию возле методов. Она парсится кодогенератором, и для нее генерируется swagger-описание. Beego генерирует документацию в формате swagger. Swagger — это открытый формат для документирования api и также набор компонентов для того, чтобы генерировать по этому описанию клиента для api и заготовку сервера. Причем это доступно для огромного количества языков, в том числе go. Конечно, писать руками определения сервисов иногда бывает заморочено, но, с другой стороны, вы сразу же получаете хорошую, красивую документацию с хорошим интерфейсом.

Если вы ищете для себя хороший старт, будь то html-сайт, так и api сервис, Beego будет неплохим выбором, учитывая то количество компонентов, которые в нем есть.

Фреймворк Gin

Рассмотрим еще один фреймворк под названием Gin. Он один из самых популярных в Go, у него очень много звезд на GitHub. Также он считается одним из самых быстрых web-фреймворков для данного языка. В частности, это достигается благодаря используемому роутеру, который мы уже рассматривали — httprouter. Благодаря ему и благодаря своим оптимизациям он работает очень быстро.

Gin — он не настолько вещь в себе, как Beego, хотя у него тоже есть свой контекст, через который происходят все операции с запросом. Например, чтобы отдать в ответе какой-то строковый параметр, мы можем использовать функцию String прямо из контекста и написать туда статус запроса и какой-то текст. Gin, помимо этого, предоставляет некоторое количество middleware, то есть, например, у него из коробки есть логгер и восстановление после паники. Благодаря тому, что он построен на хорошем роутере, он сразу может из коробки принимать параметры в роутах. У него есть встроенная basic-авторизация. Можно создавать группу роутов и на нее навешивать нужные вам цепочки middleware. Кроме того Gin встроен валидатор параметров — тоже внешняя библиотека, как и роутер, но Gin ее хорошо интегрирует в себя.

Gin — неплохой выбор, если вам нужно сделать что-нибудь маленькое и выстрое. Мы не станем подробнее останавливаться на этом фреймворке, полезнее будет самостоятельно в случае необходимости ознакомиться с документацией. В коде к лекции вы можете изучить пример простенького сервера, созданного с помощью Gin.

Логирование

Стандартный пакет log, zap, logrus

Теперь поговорим про логирование. Логирование — это очень важная часть любой программы, которая делает что-либо важное. Потому что логирование — это признаки жизни программы. Конечно, можно сказать, что, если программа ест процессор, то она работает. Однако это не всегда так. Она может уйти в какой-нибудь вечный цикл, например, и вы не сможете понять, действительно она работает или не зависла. Также без логов вы не сможете понять, правильно работает ваша программа, либо же там постоянно какие-то ошибки, и все запросы оканчиваются неудачей.

После того, как обосновали важность логирования, поговорим про несколько подходов к логированию. Мы рассмотрим стандартную библиотеку, логгер под названием ZAP и логгер под названием Logrus. Прежде для логирования мы использовали функцию Printf из пакета fmt, однако для логирования она не то чтобы очень подходит, потому что по умолчанию все выводит в std out. Поэтому логировать через fmt допустимо на этапе отладки либо в учебных примерах. Для более-менее продакшн-логирования стандартная библиотека предоставляет пакет log. На самом деле это обертка над Printf с выводом в stderr, однако есть возможность направить вывод в другое место, а также писать туда какую-нибудь дополнительную информацию, например, время, строчку и что-то еще. Также его можно завернуть в syslog — тоже из коробки.

```
// std
fmt.Printf("STD starting server at %s:%d", addr, port)
```

```
// std
log.Printf("STD starting server at %s:%d", addr, port)
```

следующий логер — ZAP — это логер, который был разработан в Google.

```
// zap
// у zap-а нет логгера по-умолчанию
zapLogger, _ := zap.NewProduction()
defer zapLogger.Sync()
zapLogger.Info("starting server",
        zap.String("logger", "ZAP"),
        zap.String("host", addr),
        zap.Int("port", port),
)
```

У него нет логгера по умолчанию, его нужно создавать. Кроме того он отличается структурным логированием. Это значит, что он не умеет делать форматированный вывод, вы должны четко указывать, какого типа параметры вы хотите туда передавать. За счет этого в ZAP'е нет аллокации памяти во время выполнения, потому что нет пустых интерфейсов, которые есть в Printf'е. В ZAP можно добавить соответственно дополнительные параметры и вывести какую-то строчку.

Последний из рассматриваемых — логер Logrus. В Logrus можно тоже дополнительно добавить дополнительные структурные поля и вывести info, либо любой другой уровень логирования. Он тоже вроде как имеет структурное логирование, но на самом деле это структурное логирование — это $map[string]interface$. Но у Logrus'а есть другие достоинства.

```
// logrus
logrus.SetFormatter(&logrus.TextFormatter{DisableColors: true})
logrus.WithFields(logrus.Fields{
        "logger": "LOGRUS",
        "host":   addr,
        "port":   port,
}).Info("Starting server")
```

Рассмотрим, как этим пользоваться на примере структуры AccessLogger, в которой будет целых три логгера: стандартный, ZAP и Logger.

```
type AccessLogger struct {
        StdLogger    *log.Logger
        ZapLogger    *zap.SugaredLogger
        LogrusLogger *logrus.Entry
}

func main(){
          AccessLogOut := new(AccessLogger)

        // std
        AccessLogOut.StdLogger = log.New(os.Stdout, "STD ", log.LUTC|log.Lshortfile)

        // zap
        sugar := zapLogger.Sugar().With(
                zap.String("mode", "[access_log]"),
                zap.String("logger", "ZAP"),
        )
        AccessLogOut.ZapLogger = sugar

        // logrus
        contextLogger := logrus.WithFields(logrus.Fields{
                "mode":   "[access_log]",
                "logger": "LOGRUS",
        })
        logrus.SetFormatter(&logrus.JSONFormatter{})
        AccessLogOut.LogrusLogger = contextLogger
```

```
        // server stuff
        siteMux := http.NewServeMux()
        siteMux.HandleFunc("/", mainPage)
        siteHandler := AccessLogOut.accessLogMiddleware(siteMux)
        http.ListenAndServe(":8080", siteHandler)
}
```

В main создаем свои логгеры в эту структуру. Новый стандартный логер, который будет писать в stdout с каким-то префиксом. Так же будет писать время в utc и будет добавлять строчку, на которой была ошибка. Это очень удобно во время отладки, однако довольно дорого, потому что нам придется пройтись в рантайме по стеку. Если логов у вас будет очень много, это грузит программу. Следующим в качестве логгера ZAP создаем отдельный логгер Sugar. Sugar — это отдельная надстройка над ZAP'ом, которая позволяет выводить какие-то данные вроде Printf. Наконец, создаем Logrus, сразу указывая, что Logrus должен форматировать все в JSON.

Теперь, чтобы работать с логерами нужен midleware. Все запросы оборачмваем в метод accessLog. Обратите внимание, раньше у в виде middleware выступали просто отдельные функции, однако оборачивать можно и в метод структуры, тогда из этого метода можно будет обращаться к каким-то отдельным полям структуры, чем мы и воспользуемся, ведь у нас в структуре целых три логера.

```
func (ac *AccessLogger) accessLogMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                start := time.Now()
                next.ServeHTTP(w, r)

                fmt.Printf("FMT [%s] %s, %s %s\n",
                        r.Method, r.RemoteAddr, r.URL.Path, time.Since(start))

                log.Printf("LOG [%s] %s, %s %s\n",
                        r.Method, r.RemoteAddr, r.URL.Path, time.Since(start))

                ac.StdLogger.Printf("[%s] %s, %s %s\n",
                        r.Method, r.RemoteAddr, r.URL.Path, time.Since(start))

                ac.ZapLogger.Info(r.URL.Path,
                        zap.String("method", r.Method),
                        zap.String("remote_addr", r.RemoteAddr),
                        zap.String("url", r.URL.Path),
                        zap.Duration("work_time", time.Since(start)),
                )

                ac.LogrusLogger.WithFields(logrus.Fields{
                        "method":      r.Method,
                        "remote_addr": r.RemoteAddr,
                        "work_time":   time.Since(start),
                }).Info(r.URL.Path)
        })
}
```

Продемонстрируем в этом методе возможности сразу всех рассматриваемых логеров. Выведем логи просто через Printf. Это очень просто делается, в одну строчку и красиво. Однако все значения будут приведены к пустому интерфейсу, что будет создавать понемножку локаций. Затем выведем логи с помощью пакета log, в общем-то это то же самое, что Printf. Затем через стандартный логгер — тот логгер, который мы указали структуре, в его выводе добавится префикс, время запросов в utc и строчка. Следующим выведем лог через ZAP. Логгируем Path, и все параметры приводим к конкретному типу данных. За счет этого нет локаций. Наконец, выводи через Logrus, в который передаем вроде бы то же самое, однако аллокации там есть.

Теперь надо запустить, проделайте это самостоятельно. Обратите внимание, логи zap выведены в виде красивого JSON'а, который очень удобно парсить какими-либо средствами, внешними программами. Можно заметить, что вроде бы ZAP и Logrus они похожи, они оба выводят много информации. В чем

между ними разница? Когда использовать один, когда другой? Logrus медленный, однако у него очень много модулей, которые отправляют в нужный приемник в нужном формате. Если вы хотите слать логи в разные места, например в Fluentd или Slack, то, возможно, логгер вам подойдет. Однако Logrus не очень производительный. Он медленнее ZAP'а в восемь раз, а локаций там почти в 30 раз больше. Поэтому если у вас логов очень-очень много и вы хотите их писать в какое-то одно место, и вам важна производительность, то, возможно, логгер ZAP будет хорошим выбором для этих целей. Ну, а если вам все равно, то вы можете использовать стандартный лог из стандартной библиотеки и не заморачиваться.

Веб-сокеты

gorilla/websocket

Поговорим про веб-сокеты. Веб-сокеты позволяют установить веб-странице полнодуплексное соединение с веб-сервером, и, таким образом, уже не только страница может выступать инициатором общения, но и веб-сервер может со своей стороны послать сразу что-то в веб. Это позволяет создавать очень интерактивные приложения, когда при появлении какого-то рода обновлений на сервере, они сразу же отправляются в веб, во фронтенд без ожидания, пока фронтенд отправит какой-либо запрос с вопросом: «А были ли какие-то обновления?»

Рассмотрим, каким образом можно работать с веб-сокетами из Go. Для начала стоит рассказать, что в Go из коробки нет поддержки веб-сокетов. Но есть библиотеки, которые реализуют полную поддержку протокола. В данном случае мы будем рассматривать библиотеку websocket из набора компонент gorilla. Итак, собственно, рассмотрим код нашего сервера.

```
package main

import (
        "encoding/json"
        "fmt"
        "html/template"
        "log"
        "net/http"
        "time"

        "github.com/gorilla/websocket"
        "github.com/icrowley/fake"
)

func main() {
        tmpl := template.Must(template.ParseFiles("index.html"))

        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
                tmpl.Execute(w, nil)
        })

        http.HandleFunc("/notifications", func(w http.ResponseWriter, r *http.Request) {
                ws, err := upgrader.Upgrade(w, r, nil)
                if err != nil {
                        log.Fatal(err)
                }
                go sendNewMsgNotifications(ws)
        })

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

У нас есть страница в корне, которая ничего не делает, кроме экспанда шаблонов. На нём мы останавливаться не будем. И страница с уведомлениями, которая занимается тем, что она апгрейдит соединение до веб-сокета, то есть веб отправляет на сервер запрос о том, можно ли сюда по веб-сокету стучаться, и мы отвечаем, что да. Отвечаем как раз при помощи команды upgrade. После этого дела то соединение,

мы уже в отдельной корутине мы начинаем обрабатывать. В случае, если у нас какой-либо синхронный язык, например, PHP, то там с веб-сокетом работать не то что бы удобно. Дорого. Веб-сокеты хорошо себя проявляют при асинхронном подходе. В JavaScript'е, например, ну и в Go, конечно же, тоже. Мы можем держать очень большое количество веб-сокет соединений до тех пор, пока у нас хватает памяти. И при появлении какого рода обновлений сразу слать туда данные. В данном случае я буду каждые 3 секунды слать новое сообщение туда.

```
func sendNewMsgNotifications(client *websocket.Conn) {
        ticker := time.NewTicker(3 * time.Second)
        for {
                w, err := client.NextWriter(websocket.TextMessage)
                if err != nil {
                        ticker.Stop()
                        break
                }

                msg := newMessage()
                w.Write(msg)
                w.Close()

                <-ticker.C
        }
}
```

То есть мы получаем новый пакет с данными, NewWriter. Помимо Writer'а там может ещё быть ping, pong, сообщение о закрытии соединения. Либо же чтение. В данном случае будем в веб-сокет что-то записывать. Если у записать не получилось, останавливаем ticker и выходим из цикла. Ticker надо останавливать, чтобы не было утечек памяти. После этого я получаем новое сообщение, там какие-то фейковые данные, которые пишем в w.Write(), полученный от клиента, сразу же это сообщение закрываем. Оно уходит на сервер, и мы ждем следующего сообщения. Собственно, вот и всё, как можно организовать отправку данных с сервера на клиент.

Работа с кокетами очень удобна для разного рода нагруженных сервисов, где много клиентов, где часто приходят любого рода уведомления. Сайт становится действительно очень интерактивным. Вам пришло письмо, вы сразу же об этом получаете сообщение, оно сразу же появляется у вас на странице. Веб-сокет иметь в целом дешевле, чем организовывать регулярный polling, то есть постоянный запрос с клиента на сервер: «А было ли обновление? А было ли обновление?» Потому что вам придётся больше информации в итоге гонять. А с веб-сокетами вам с сервера может прийти сообщение тогда, когда оно нужно.

Шаблонизация

Компилируемые шаблоны

В предыдущем курсе мы рассматривали шаблонизатор, который поставляется вместе со стандартной библиотекой в Go. Этот шаблонизатор обладает довольно большими возможностями, в нем есть фактически все, что нужно для полноценной работы. Однако у него есть один недостаток. Для некоторых он может быть весьма серьезным. Весь этот шаблонизатор построен на рефлекте, то есть все его вычисления происходят не во время компиляции, а в runtime'е. Если ваше приложение интенсивно отдает шаблоны, то, возможно, вы захотите это место у себя оптимизировать. Мы рассмотрим шаблонизатор Hero, один из самых быстрых, который из шаблонов компилирует Go-код.

Давайте посмотрим, как его использовать в коде.

```
package main

import (
        "bytes"
        "fmt"
        "net/http"

        "coursera/template_adv/item"
        "coursera/template_adv/template"
```

```
)

//go:generate hero -source=./template/

var ExampleItems = []*item.Item{
        &item.Item{1, "rvasily", "Mail.ru Group"},
        &item.Item{2, "username", "freelancer"},
}

func main() {
        http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
                buffer := new(bytes.Buffer)
                template.Index(ExampleItems, buffer)
                w.Write(buffer.Bytes())
        })

        fmt.Println("starting server at :8080")
        http.ListenAndServe(":8080", nil)
}
```

Вот пакет main, импортируем item'ы, не будум на них останавливаться, там всего одна структура, шаблоны, к ним вернемся чуть позже. Далее у нас есть некоторый набор данных, и мы вызываем из шаблонов функцию Index. Обратите внимание, мы не указываем, какие файлы надо парсить, не вызываем другие шаблонизаторы, всего лишь подключили какой-то свой пакет и вызываем оттуда функцию, передавая ей байтовый буфер, чтобы писать туда данные.

Все самое интересное происходит в пакете template. Собственно, как получается сам шаблон? У нас есть базовая обертка над шаблоном, (наследование шаблонов — это удобно). Обратимся, собственно, к шаблону с логикой.

```
<%!
import (
    "coursera/template_adv/item"
)
%>

<%: func Index(items []*item.Item, buffer *bytes.Buffer) %>

<%~ "base.html" %>

<%@ body { %>
<div class="container">
    <h1>Posts</h1>
    <table class="table">
    <thead>
        <tr>
        <th>#</th>
        <th>Title</th>
        <th>Description</th>
        <th style="width:140px;"><a href="/items/new" class="btn btn-success">New</a></th>
        </tr>
    </thead>
    <tbody>
    <% for _, item := range items { %>
    <tr>
        <td><%==i item.Id %></td>
        <td><%= item.Title %></td>
        <td><%= item.Description %></td>
        <td>
            <a href="/items/<%==i item.Id %>" class="btn btn-primary">Edit</a>
            <span data-id="<%==i item.Id %>" class="do-delete btn btn-danger">Del</span>
        </td>
```

```
        </tr>
        <% } %>
        </tbody>
    </div>
    <% } %>
```

Что мы тут видим? Для начала мы видим import. Это нужно для того, чтобы шаблонизатор смог скомпилировать html с псевдоразметкой в код на Go. Далее мы объявляем функцию (опять-таки, что странно), мы указываем, что мы будем наследоваться от base.html, определяем блок кода, который хотим заменить, и тут мы уже пишем html-цикл. Обратите внимание, это не какой-то там цикл, который цикл встроен в шаблонизатор. Он опять-таки очень похож на цикл на Go. В цикле мы выводим переменные, они экранируются. Для целочисленных типов, таких, как id, мы должны указать, к чему привести эту переменную, чтобы шаблонизатор подставил нужную функцию преобразования. Это похоже ранний php, когда отдельные шаблонизаторы еще не были в моде.

Теперь этот файл нужно скомпилировать в код на Go. Делается это из папки с main'ом при помощь команды компиляции шаблонов

```
hero -source=./template/
```

Однако, руками обычно такое не запускают, потому что может быть много разных блоков, много разных файлов, которые распаковывают свои шаблоны, и все их компилировать не очень удобно. Поэтому Go есть специальный служебный комментарий go:generate, обратите внимание, он присутствует в пакете main. После того, как вы в таком комментарии указали, что вы хотите выполнить, например, компиляцию шаблонов с помощью hero, останется только запустить специальную программу go generate, которая найдет все такие комментарии и выполнит ту программу, которая в них есть. В коде с урока лежит уже скомпилированный шаблон, однако попробуйте проделать это самостоятельно.

Еще момент, который стоит упомянуть. Мы используем простой буфер данных, и каждый раз его создаем, потом пишем что-то в ответ, затем он уничтожается — уходит на heap и собирается сборщиком мусора. Однако мы можем использовать sync.Pool и переиспользовать буфер. Таким образом, вы не будете грузить garbage collector, у вас будут паузы на сборку мусора гораздо меньше. Если ваши шаблоны с html весят по несколько мегабайт, то такого рода переиспользование памяти становится необходимым для работы под большой нагрузкой.

Напоследок обсудим недостатки компиилируемых шаблонов. Если вы используете шаблонизатор из стандартной библиотеки, либо любой другой, который работает в runtime'е, то вы можете шаблоны перезагрузить во время работы вашей программы. Если в вашем сервисе frontend разработка отделена от backend разработки, то вам иногда бывает удобно сделать релиз html-части отдельно от backend'а. И ваша программа может в runtime'е перезагрузить шаблоны, и они обновятся. В случае со скомпилированными шаблонами такое не пройдет. Потому что ваш шаблон — это функция на Go. Для того чтобы переразложить такие шаблоны, вам придется перекомпилировать всю вашу программу. Это главный недостаток компилируемых шаблонов, зато он компенсируется скоростью.

Управление зависимостями

dep, glide, gb

В этом разделе поговорим про организацию зависимостей, то есть внешних пакетов с кодом, которые требуются вашему приложению. Во многих других языках зависимости устанавливаются через разного рода менеджеры пакетов, либо как какие-то внешние пакеты, например, rmp в Node.js, pip в Python либо RubyGems. В Go тоже есть встроенный инструмент для подтягивания внешних пакетов — go get. Однако есть и большой нюанс: он устанавливает все зависимости в одну папку GOPATH. Это может быть критично, если вы работаете над несколькими проектами. У вас могут отличаться зависимости, которые нужны для этих проектов. Например, где-то нужны новые возможности, но сломана обратная совместимость, и вы не хотите тратить время на починку старого проекта после установки нового пакета.

Как быть, что делать? Есть первый вариант — лоб — менять GOPATH. То есть вы можете в переменной окружения просто переустановить значение и новый пакет устанавливать в другое место. Но этот подход не очень переносим и требует для удобства какой-нибудь Makefile. Поэтому в рабочей группе Go работают над новыми инструментами. Пока они работают, уже родилось несколько менеджеров зависимостей, которые ставят все зависимости в папку vendor (работа с зависимостями в Go называется vendoring).

Мы обзорно рассмотрим три менеджера зависимостей — glide, dep, gb Изучите код с урока. Там есть по папке для каждого менеджера. В каждой папке лежит readMe-file, с основными командами, которые поддерживает тот или иной менеджер, и код, который не должен собраться, потому что ни в GOROOT, ни в GOPATH не установлены нужные пакеты.

Glide — первый менеджер зависимостей, про который мы поговорим. С помощью команды glide create он может просканировать ваш ваш код на предмет тех зависимостей, которые там используются, и построить glide-файл в формате yaml, в котором будут все зависимости. По этому файлу можно сделать установку нужных зависимостей и подзависимостей: либо через glide get, либо через glide install, который поставит все зависимости в папку vendor. Дело в том, что в Go пришли к такому решению, что, если рядом с вашим кодом, который вы запускаете, есть папка vendor, то все зависимости будут искаться в первую очередь в этой папке, а не в GOROOT и GOPATH. После установки пакетов также появится файл glide.lock, в котором будет вся информация про зависимости.Есть нюансы — подпакеты подтянуть не получится, если вы не укажете полный путь.

Следующий менеджер, который мы рассмотрим, называется dep. dep — это будущий стандартный менеджер зависимостей, сейчас он находится в зачаточном состоянии, то есть он умеет некоторые вещи, но не все. Функционал его похож на glide. У него есть функция dep init, которая тоже просканирует ваш код на предмет того, какие зависимости стоит установить, и сразу же установит их так же в папку vendor. Он так же хранит список зависимостей, так же хранит версии этих зависимостей. Функционал в нем пока не ососбо широк, но это будущий стандартный менеджер зависимостей и знать по него стоит.

И еще один менеджер, который мы рассмотрим, называется gb. Он отличается от всех предыдущих тем, что внутри себя перекрывает GOPATH и позволяет вам работать с каждым из ваших проектов действительно полноценно в отдельной директории. Для этого у него есть своя команда gb build. Так же он позволяет импортировать подпакеты, не прописывая полные пути до них. В связи с этим gb бывает удобен для совсем изолированных сервисов, которые вы не хотите класть в свой GOPATH. Однако этот менеджер не так активно развивается, в нем нет возможности подтянуть все зависимости автоматически , например, пакеты из защищенных источников, когда репозиторий закрыт сертификатом, например. Их придется подтягивать руками руками. К тому же он достаточно медленный.

Конечно, есть и другие менеджеры, это не полный список. Кому-то понравится glide, кому-то dep, кто-то выберет gb, кому-то придется по душе govendor или какой-то другой менеджер, который мы не упомянули.