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

SQL

database/sql и mysql

В этой главе рассматрим, как работать с SQL базами данных и с Go. Все в данном уроке будет реализовано через MySQL, для Postgres отличия будут очень незначительные, и файлы для него вы можете найти в файлах к уроку.

Итак, сначала посмотрим на схему данных.

```
SET NAMES utf8;
SET time_zone = '+00:00';
SET foreign_key_checks = 0;
SET sql_mode = 'NO_AUTO_VALUE_ON_ZERO';

DROP TABLE IF EXISTS `items`;
CREATE TABLE `items` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(255) NOT NULL,
  `description` text NOT NULL,
  `updated` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `items` (`id`, `title`, `description`, `updated`) VALUES
(1,        'database/sql',        'Рассказать про базы данных',        'rvasily'),
(2,        'memcache', 'Рассказать про мемкеш с примером использования', NULL);
```

У нас есть таблица items, в ней поля id, title, desctiption и поле updated. Обратите внимание, что поле updated у нас может быть NULL. Тут же создадаем таблицу и вставляем несколько элементов. Теперь просмотрим, каким образом начать работать с базой.

В стандартной библиотеке Go нету встроенного драйвера, однако авторы стандартной библиотеки поступили очень мудро и сделали интерфейс. Называется он database/sql. Этот интерфейс могут реализовывать уже разные базы данных, за счет чего вы сможете более или менее одинаково работать с MySQL, Postgres, Oracle, SQL и какими-нибудь другими базами данных. Мы будем использовать драйвер MySQL. Обратите внимание: подключать его надо использованием пустой переменной. Дело в том, что сам драйвер не экспортирует никакие функции, потому что все идет через database/sql. Однако драйвер внутри себя через функцию init регистрирует обработчик MySQL, который будет вызван далее и для этого нужен импорт, но с подчеркиванием, чтобы компилятор Go не ругался на неиспользуемый пакет.

```
package main

import (
```

```
        "database/sql"
        "fmt"
        "html/template"
        "net/http"
        "strconv"

        "github.com/gorilla/mux"

        _ "github.com/go-sql-driver/mysql"
)

type Item struct {
        Id          int
        Title       string
        Description string
        Updated     sql.NullString
}

type Handler struct {
        DB   *sql.DB
        Tmpl *template.Template
}

func main() {

        // основные настройки к базе
        dsn := "root@tcp(localhost:3306)/coursera?"
        // указываем кодировку
        dsn += "&charset=utf8"
        // отказываемся от prapared statements
        // параметры подставляются сразу
        dsn += "&interpolateParams=true"

        db, err := sql.Open("mysql", dsn)

        db.SetMaxOpenConns(10)

        err = db.Ping() // вот тут будет первое подключение к базе
        if err != nil {
                panic(err)
        }

        handlers := &Handler{
                DB:   db,
                Tmpl: template.Must(template.ParseGlob("../crud_templates/*")),
        }

        // в целям упрощения примера пропущена авторизация и csrf
        r := mux.NewRouter()
        r.HandleFunc("/", handlers.List).Methods("GET")
        r.HandleFunc("/items", handlers.List).Methods("GET")
        r.HandleFunc("/items/new", handlers.AddForm).Methods("GET")
        r.HandleFunc("/items/new", handlers.Add).Methods("POST")
        r.HandleFunc("/items/{id}", handlers.Edit).Methods("GET")
        r.HandleFunc("/items/{id}", handlers.Update).Methods("POST")
        r.HandleFunc("/items/{id}", handlers.Delete).Methods("DELETE")

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

```
// не используйте такой код в прошакшене
// ошибка должна всегда явно обрабатываться!
func __err_panic(err error) {
        if err != nil {
                panic(err)
        }
}
```

Рассматривать мы будем наш item, который мапится из строчки MySQL в структуру с полями id, title, description и полем updated, которое представлено каким-то полем NullString. Подключение к базе осуществляется через формат dsm. При подключении надо указать имя пользователя, возможно пароль, потом формат, по которому мы подключаемся — может быть TCP, может быть UNIX-сокет, адрес сервера и имя базы данных. Затем указываем уже в параметрах — по сути, это query-параметры — кодировку и параметр interpolateParams, который отключит не prepare statement'ы, а будет сразу подставлять параметры. Теперь подключаемся — обратите внимание, мы используем функцию SQL из пакета database/sql, там используем Open и только в параметрах указываем конкретную базу — mysql. Следующей строкой устанавливаем максимальное число подключений. Дело в том, что MySQL имеет полностью синхронный протокол, то есть пока один запрос не отработал, вы не можете отправлять туда никакой другой. Для того чтобы в Go с этим работать и отправлять много запросов, внутри создается несколько подключений, как раз количество этих подключений мы определили. Затем надо выполнить пингование базы данных, то есть подключиться. sql.Open не создает подключения сразу, в только инициализирует объект, а уже db.Ping подключается к базе данных. Далее для удобной обработки запросов используем Gorilla mux-роутер, Создаем handler, туда передаем базу данных и шаблоны, с которыми будем работать, чтобы не использовать Далее идет ргистрация хендлеров — тут ничего сложного нету, какие-то методы доступны через get, какие-то — через post, и один — через delete.

Рассмотрим возможные операции с базой. Как сделать select в базу?

```
func (h *Handler) List(w http.ResponseWriter, r *http.Request) {

        items := []*Item{}

        rows, err := h.DB.Query("SELECT id, title, updated FROM items")
        __err_panic(err)
        for rows.Next() {
                post := &Item{}
                err = rows.Scan(&post.Id, &post.Title, &post.Updated)
                __err_panic(err)
                items = append(items, post)
        }
        // надо закрывать соединение, иначе будет течь
        rows.Close()

        err = h.Tmpl.ExecuteTemplate(w, "index.html", struct {
                Items []*Item
        }{
                Items: items,
        })
        if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
}
```

Для этого мы обращаемся к объекту DB в хендлере и выполняем метод Query, который принимает в себя первым аргументом SQL-запрос, в котором могут быть плейсхолдеры, и следующими аргументами — уже значения для плейсхолдеров. Нам вернутся строки, удовлетворяющие запросу, мы по ним итерируемся. Остается только распаковать rows из бинарного формата MySQL, для этого есть функция Scan, в которую надо передать параметры, в которые будет распаковано значение. Соответственно, значение должно быть подходящего типа. Id — это int, Title — это string, а вот Updated, Updated — он у нас имеет тип NullString. Зачем нужен тип NullString? Поле Updated может иметь внутри nil. И если nil попробовать присвоить в строку, будет ошибка. Соответственно, нужен способ, каким образом можно проверять, есть ли там

значение NULL или нет. Для этого в пакете database/sql есть NullString, NullInt и еще несколько полей, которые имеют поле Valid, в котором находится булевый флаг — там NULL или не NULL, — и уже String, который нам будет говорить, какое там реальное значение. Можно, конечно, вместо этого использовать указатель. Когда полностью вычитали все записи из запроса, его обязательно нужно его закрыть, чтобы избежать утечки памяти.

Теперь посмотрим, как добавить запись.

```
func (h *Handler) Add(w http.ResponseWriter, r *http.Request) {
        // в целям упрощения примера пропущена валидация
        result, err := h.DB.Exec(
                "INSERT INTO items (`title`, `description`) VALUES (?, ?)",
                r.FormValue("title"),
                r.FormValue("description"),
        )
        __err_panic(err)

        affected, err := result.RowsAffected()
        __err_panic(err)
        lastID, err := result.LastInsertId()
        __err_panic(err)

        fmt.Println("Insert - RowsAffected", affected, "LastInsertId: ", lastID)

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

Через функцию query сделать insert не получится, она только возвращает данные. Для работы с insert, update есть функция exec. Туда передаем sql-запрос. Обратите внимание, тут уже используем placeholder знак вопроса. Значение туда будет сразу поставлено с нужным эскейпингом. В данном примере мы не делаем никакой ни валидации, ни авторизации, ни защиты от серф-токенов, вам они, конечно же, в продакшене будут нужны.

Теперь редактирование записи.

```
func (h *Handler) Edit(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        id, err := strconv.Atoi(vars["id"])
        __err_panic(err)

        post := &Item{}
        // QueryRow сам закрывает коннект
        row := h.DB.QueryRow("SELECT id, title, updated, description FROM
                                                        items WHERE id = ?", id)

        err = row.Scan(&post.Id, &post.Title, &post.Updated, &post.Description)
        __err_panic(err)

        err = h.Tmpl.ExecuteTemplate(w, "edit.html", post)
        if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
}
```

Здесь уже нужны две функции. При помощи одной мы отобразим пользователю запись, которую он хочет редактирвать, а при помощь второй внесем его изменения в базую. Выбираем запись, чтобы отобразить её в вебе, используя уже не query, а QueryRow. QueryRow сам выберет нужную запись, закроет connect, то есть QueryRow нужен для работы с единичными записями. Теперь сам update.

```
 func (h *Handler) Update(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        id, err := strconv.Atoi(vars["id"])
```

```
        __err_panic(err)

        // в целям упрощения примера пропущена валидация
        result, err := h.DB.Exec(
                "UPDATE items SET"+
                        "`title` = ?"+
                        ",`description` = ?"+
                        ",`updated` = ?"+
                        "WHERE id = ?",
                r.FormValue("title"),
                r.FormValue("description"),
                "rvasily",
                id,
        )
        __err_panic(err)

        affected, err := result.RowsAffected()
        __err_panic(err)

        fmt.Println("Update - RowsAffected", affected)

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

В апдейте тоже конструируем sql-запрос: update, title, description updated. Точно так же берем значения сразу из формы без валидации (не делайте так). И для каждого поля прописываем updates на какое-то статичное значение. Последним параметром идет Id записи, которую мы меняем. С помощью RowsAffected можно посмотреть, сколько записей было изменено этим запросом.

В удалении тоже все просто.

```
func (h *Handler) Delete(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        id, err := strconv.Atoi(vars["id"])
        __err_panic(err)

        result, err := h.DB.Exec(
                "DELETE FROM items WHERE id = ?",
                id,
        )
        __err_panic(err)

        affected, err := result.RowsAffected()
        __err_panic(err)

        fmt.Println("Delete - RowsAffected", affected)

        w.Header().Set("Content-type", "application/json")
        resp := `{"affected": ` + strconv.Itoa(int(affected)) + `}`
        w.Write([]byte(resp))
}
```

Передаем sql-запрос и один параметр. Обратите внимание, нигде не используем ручное добавление, конкатенацию самих параметров в базу данных. Всегда пользуйтесь placeholder. Если вы попробуете через sprintf или еще как-то руками конструировать запросы, где-то вы проглядите и будет sql-инъекция. Sql-инъекция — это неприятно, и это очень частая причина взломов ваших сайтов, поэтому placeholder, placeholder, placeholder.

Иногда бывает неудобно руками распаковывать все, как мы это делали в примерах. И для этого есть специальные средства. Про них мы поговорим далее.

GORM - применение reflect'а для SQL

В предыдущем разделе мы рассматривали, каким образом можно руками работать с SQL. Иногда это бывает очень неудобно, особенно когда приходится выполнять множество однотипных простых операций с базой. Поэтому возникает желание использовать какую-то обертку, которая будет при помощи какой-то "магии"скрывать от нас всю внутреннюю работу за одной простой одну функцией. Для этого в Go есть специальные внешние библиотеки. Работают они либо через рефлексию (получается гибче, но с небольшим overhead'ом), либо через кодогенерацию (быстрее, но с необходимостью пересобирать проект на каждое изменение).

В этом разделе мы рассмотрим библиотеку gorm, которая как оборачивает всю работу SQL в себя за счет рефлексии. Опять рассмотрим все основные операции на примерах. Начнем с подключения.

```
package main

import (
        "fmt"
        "html/template"
        "net/http"
        "strconv"

        "github.com/gorilla/mux"
        "github.com/jinzhu/gorm"

        _ "github.com/go-sql-driver/mysql"
)

func main() {

        // основные настройки к базе
        dsn := "root@tcp(localhost:3306)/coursera?"
        // указываем кодировку
        dsn += "&charset=utf8"
        // отказываемся от prapared statements
        // параметры подставляются сразу
        dsn += "&interpolateParams=true"

        db, err := gorm.Open("mysql", dsn)
        db.DB()
        db.DB().Ping()
        ...
```

После основных настроек подключения к базе, вызываем gorm.Open, который подключается через MySQL. Внутри себя он создает свою базу. Дальше подключаемся — пингуем — уже базу gorm'а. То есть в создании базы изменений не очень много — изменили одну строчку на другую.

Теперь рассмотрим, как выбирать записи и с ними работать. Сначала выбираем.

```
func (h *Handler) List(w http.ResponseWriter, r *http.Request) {

        items := []*Item{}

        db := h.DB.Find(&items)
        err := db.Error
        __err_panic(err)

        err = h.Tmpl.ExecuteTemplate(w, "index.html", struct {
                Items []*Item
        }{
                Items: items,
        })
        if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
```

```
                return
        }
}
```

Как и в прошлый раз хотим найти просто все записи. Для этого вызываем метод Find нашей базы, передавай туда слайс структур, куда запиется результат. Если раньше нужно было писать SQL-запрос, что-то выбирать, потом распаковывать это руками, то теперь есть одна простая функция. Правда, она выберет все, и если у вас есть какие-то очень тяжелые поля, придется ставить какие-то ограничения.

Создание записи.

```
func (h *Handler) Add(w http.ResponseWriter, r *http.Request) {

        newItem := &Item{
                Title:       r.FormValue("title"),
                Description: r.FormValue("description"),
        }
        db := h.DB.Create(&newItem)
        err := db.Error
        __err_panic(err)
        affected := db.RowsAffected

        fmt.Println("Insert - RowsAffected", affected, "LastInsertId: ", newItem.Id)

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

Снова все предельно просто — создаем структуру и вызываем метод Create, куда передаем новосозданную структуру.

Чтобы, как в предыдущем примере получить RowsAffected достаточно вызвать соответствующую функцию, а LastId сразу же запишется в нужное поле вставляемой структуры. Это произойдет за счет тега структуры. Мы помечаем поле Id SQL-тегом автоикремент, кроме того указываем, что для горма это *primary_key*. в это поле LastId записать нужно.

```
type Item struct {
        Id          int `sql:"AUTO_INCREMENT" gorm:"primary_key"`
        Title       string
        Description string
        Updated     string `sql:"null"`
}
```

Еще сразу обратим внимание, что Updated помечено тегом, говоряшим, что поле может быть null. Кроме тегов есть еще много разных ыозможностей. Напримре, переопределить таблицу для структуры, или навешать разные триггеры, например, триггер, вызывающийся при пересохранении результата.

```
func (i *Item) TableName() string {
        return "items"
}

func (i *Item) BeforeSave() (err error) {
        fmt.Println("trigger on before save")
        return
}
```

Но не будем долго на этом останавливаться.

Пойдем дальше. Редактирование. Опять же оно разбито на две функции.

```
func (h *Handler) Edit(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        id, err := strconv.Atoi(vars["id"])
        __err_panic(err)
```

```
        post := &Item{}

        db := h.DB.Find(post, id)
        err = db.Error
        if err == gorm.ErrRecordNotFound {
                fmt.Println("Record not found", id)
        } else {
                __err_panic(err)
        }

        err = h.Tmpl.ExecuteTemplate(w, "edit.html", post)
        if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
}
```

Найти нужную запись можно с помощью функции Find, указав ей, куда записать результат и какой Id хотим найти. То есть раньше я руками указывали все поля, доставали, распаковывали. Теперь — всего одна функция, да проверка ошибок. Так же просто все стало в Update.

```
func (h *Handler) Update(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        id, err := strconv.Atoi(vars["id"])
        __err_panic(err)

        post := &Item{}
        h.DB.Find(post, id)

        post.Title = r.FormValue("title")
        post.Description = r.FormValue("description")
        post.Updated = "rvasily"

        db := h.DB.Save(post)
        err = db.Error
        __err_panic(err)
        affected := db.RowsAffected

        fmt.Println("Update - RowsAffected", affected)

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

Сначала достаем нужную запись, чтобы какие-то поля не перезаписать, потом обновляем нужные поля, и вызываем функцию Save.

Для удаление тоже есть отдельная функция Delete, куда передаем структуру, (уже не единичное полеи не просто идентификатор), чтобы gorm знал, вообще из какой таблицы это удалять.

```
func (h *Handler) Delete(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        id, err := strconv.Atoi(vars["id"])
        __err_panic(err)

        db := h.DB.Delete(&Item{Id: id})
        err = db.Error
        __err_panic(err)
        affected := db.RowsAffected

        fmt.Println("Delete - RowsAffected", affected)

        w.Header().Set("Content-type", "application/json")
        resp := `{"affected": ` + strconv.Itoa(int(affected)) + `}`
```

```
        w.Write([]byte(resp))
}
```

Мы рассмотрели самые простые операции. Как видно все они пишутся очень и очень простою Кроме того у горма возможностей гораздо больше. Он может и делать джойны, и валидацию проверять, и триггеров у него много. Хотя за это и приходится платить небольшим overhead'ом на рефлексию.

Если все так просто и так много возможностей, почему query-билдеры вообще не заполонили весь мир и мы еще пишем иногда SQL-запрос руками? Дело в том, что SQL очень гибок, в нем можно делать подзапросы, или какие-то очень хитрые джойны, не всегда это получается сделать через обертку.

sql-injection

В этос разделе рассмотри вопрос, который не имеет непосредственного отношения к нашему курсу, но который обязательно должен быть затронут — sql-инъекцию, поскольку это самая распространенная уязвимость в современных веб-приложениях.

Представьте, что у нас есть какя-то форма регистрации, куда нужно вводить логин, пароль. Давайте рассмотрим слудующий код.

```
// ПЛОХО! НЕ ДЕЛАЙТЕ ТАК!
// параметры не экранированы должным образом
// мы подставляем в запрос параметр как есть
query := fmt.Sprintf("SELECT id, login FROM users WHERE login = '%s' LIMIT 1", inputLogin)

body += fmt.Sprintln("Sprint query:", query)

row := db.QueryRow(query)
err := row.Scan(&id, &login)

if err == sql.ErrNoRows {
        body += fmt.Sprintln("Sprint case: NOT FOUND")
} else {
        PanicOnErr(err)
        body += fmt.Sprintln("Sprint case: id:", id, "login:", login)
}
```

В нем мы поставляем параметр руками в запрос и не экранируем параметры должным образом. Стоит ввести пользователя, например, 404' or login = 'admin, как запрос, который мы отправляем в базу превратится в SELECT id, login FROM users WHERE login = '404' or login = 'admin' и хитрый пользователь в данном случае получит информацию об админской учетке. Это и называется sql-инъекция. В варианте же с плейсхолдерами такого не произойдет.

```
// ПРАВИЛЬНО
// Мы используем плейсхолдеры, там параметры будет экранирован должным образом
row = db.QueryRow("SELECT id, login FROM users WHERE login = ? LIMIT 1", inputLogin)
err = row.Scan(&id, &login)
if err == sql.ErrNoRows {
        body += fmt.Sprintln("Placeholders case: NOT FOUND")
} else {
        PanicOnErr(err)
        body += fmt.Sprintln("Placeholders id:", id, "login:", login)
}

w.Write([]byte(body))
```

Запомните, что никогда нельзя подставлять значение в запрос без экранирования. В базах данных почти везде поддерживаются плейсхолдеры. В mysql это символ знак вопроса. Мы составляем предварительный запрос, если он нужен, с плейсхолдерами, а сам параметр, который мы хотим подставить вместо знака вопроса, передаем аргументом в QueryRow. QueryRow — вариатик функция, она принимает любое количество параметров, и она принимает пустой интерфейс, поэтому можно передать любое значение.

Key/value

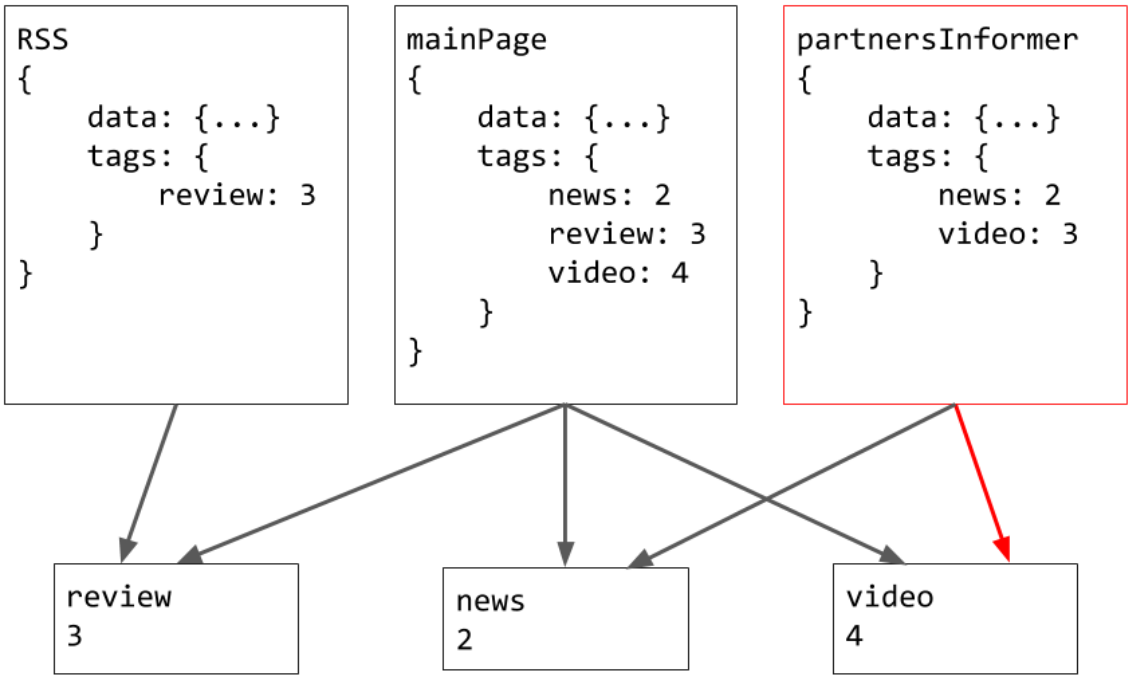
Memcached - пример с тегированным кешем

В этой главе мы рассмотрим, каким образом можно работать с memcache из го. memcache — это очень простое key-value хранилище. У него не очень много возможностей, однако это компенсируется простой и надежностью. На самом деле, рассматривать просто сохранение значений memcache и получение оттуда данных — это примерно так же неинтересно, как и рассматривать присвоение значений в переменные. Поэтому пройдемся очень быстро и перейдем к более практическому примеру.

```
package main

import (
        "fmt"
        "github.com/bradfitz/gomemcache/memcache"
)

func main() {
        MemcachedAddresses := []string{"127.0.0.1:11211"}
        memcacheClient := memcache.New(MemcachedAddresses...)

        mkey := "coursera"

        memcacheClient.Set(&memcache.Item{
                Key:        mkey,
                Value:      []byte("1"),
                Expiration: 3,
        })

        memcacheClient.Increment("habrTag", 1)

        item, err := memcacheClient.Get(mkey)
        if err != nil && err != memcache.ErrCacheMiss {
                fmt.Println("MC error", err)
        }

        fmt.Printf("mc value %#v\n", item)

        memcacheClient.Delete(mkey)

        item, err = memcacheClient.Get(mkey)
        if err == memcache.ErrCacheMiss {
                fmt.Println("record not found in MC")
        }

}
```

Итак, команда Set сохраняет запись, независимо от того была она или нет. Если была, происходит перезапись. Для этого нужно передать полностью всю запись memcache Item, ключ — строку, Value — слайс-байт, и Expiration — время в секундах, через которое запись будет неактуальна. Increment увеличивает запись на 1 или на -1, в зависимости от того что вы передадите. Get получает запись по ключу. Обратите внимание, что если запись не найдена, признак об этом вернется вам в специальном коде ошибки. В удалении тоже ничего сложного и интересного.

Не будем на этом останавливаться и пойдем дальше. Memcache очень простой, но хочется делать более сложное кеширование. Рассмотрим следующую схему.



Представьте, что у нас контент-проект и есть много различных закешированных данных, например RSS, главная страница, какой-то информер для партнеров. При этом если вы будете пытаться каждую из этих запись — RSS, mainPage, partnersInformer — сохранять в админке, то, для того чтобы сбросить, когда у вас что-то новое добавляется, вы быстро утомитесь и наверняка что-то пропустите. Поэтому есть подход, который называется «тегированный кеш», когда вы храните не сразу свои данные, а еще какую-то небольшую метаинформацию в нем. Например, теги. Это один из вариантов такой инвалидации кеша, когда у вас неизвестно количество самих закешированных данных.

Теги — это такая мапа, string, int в данном случае, где ключ указывает на какую-то другую запись в memcache, а значение — это какой-то числовой код, например timestamp, время. И если вы увеличите время, инкрементируете счетчик в соответствующей тегу записи memcache, используя команду Increment, либо просто поставите туда текущее значение Unix timestamp, то про все записи, у которых есть такой тег, когда вы их достанете из кеша, распакуете, вы сможете понять, что они уже не валидны и кеш нужно перестроить. На картинке, например, partnersInformer выделен красным, потому что тег video у него равен 3, то есть запись была закеширована, когда тег video был равен 3. Однако текущее значение у тега video — 4. Это значит, что кеш уже не валиден, и его нужно перестроить. При этом, изменив значение тега в админке, вы сразу же не вызываете каскадное перестроение всего, перестраиваться кеш будет по мере запроса.

Давайте посмотрим, как реализовать такую схему, используя Go. Пример будет довольно большой, если у вас закипят мозги, знайте — так и задумано.

Для начала сделаем структуру.

```
type TCache struct {
        *memcache.Client
}
```

Она включает в себя клиент memcache, то есть у её экземпляров мы сможем вызывать все методы, которые предоставляет memcache.Client.

Рассмотрим, что происходит в main'е.

```
func main() {
        MemcachedAddresses := []string{"127.0.0.1:11211"}
        memcacheClient := memcache.New(MemcachedAddresses...)

        tc := &TCache{memcacheClient}

        mkey := "habrposts"
        tc.Delete(mkey)

        rebuild := func() (interface{}, []string, error) {
                habrPosts, err := GetHabrPosts()
                if err != nil {
                        return nil, nil, err
                }
                return habrPosts, []string{"habrTag", "geektimes"}, nil
        }

        fmt.Println("\nTGet call #1")
        posts := RSS{}
        err := tc.TGet(mkey, 30, &posts, rebuild)
        fmt.Println("#1", len(posts.Items), "err:", err)

        fmt.Println("\nTGet call #2")
        posts = RSS{}
        err = tc.TGet(mkey, 30, &posts, rebuild)
        fmt.Println("#2", len(posts.Items), "err:", err)

        fmt.Println("\ninc tag habrTag")
        tc.Increment("habrTag", 1)

        go func() {
                // time.Sleep(time.Millisecond)
                fmt.Println("\nTGet call #async")
                posts = RSS{}
                err = tc.TGet(mkey, 30, &posts, rebuild)
                fmt.Println("#async", len(posts.Items), "err:", err)
        }()

        fmt.Println("\nTGet call #3")
        posts = RSS{}
        err = tc.TGet(mkey, 30, &posts, rebuild)
        fmt.Println("#3", len(posts.Items), "err:", err)
}
```

Инициализируем нашу структуру — tc — так я будем называть кеш. И начинаем с того, что я грохнем старые данные, которые были закешированы, чтобы пример у нас был чистым. Для этого вызываем Delete — метод memcache.Client, Далее создаем функцию rebuild, подробнее её рассмотри чуть позднее. Теперь как выглядит интрефейс по тегированному кешу? Вызываем функцию TGet от ключа с кешом. времени жизни ключа (у нас это 30 секунд), переменной, куда надо записать RSS значение и передаем функцию rebuild. Теперь пришло время посмотреть внимательней, что она делает и когда она будет вызвана. Эта функция получает значение уже из базы данных или какого-то другого хранилища. В данном примере она идет на сайт «Хабра» в RSS и получает оттуда посты. Возвращает она interface, слайс стрингов и ошибку. interface нужен, для того чтобы механизм возврата был универсальным. Слайс стрингов — это те теги, которые мы хотим сохранить для данного кеша. Да, это анонимная функция. И мы передаем ее в TGet.

Как устроена функция TGet?

```
func (tc *TCache) TGet(
        mkey string,
        ttl int32,
        in interface{},
```

```
        rebuildCb RebuildFunc,
) (err error) {
        inKind := reflect.ValueOf(in).Kind()
        if inKind != reflect.Ptr {
                return fmt.Errorf("in must be ptr, got %s", inKind)
        }

        tc.checkLock(mkey)

        itemRaw, err := tc.Get(mkey)
        if err == memcache.ErrCacheMiss {
                fmt.Println("Record not found in memcache")
                return tc.rebuild(mkey, ttl, in, rebuildCb)
        } else if err != nil {
                return err
        }

        item := &CacheItem{}
        err = json.Unmarshal(itemRaw.Value, &item)
        if err != nil {
                return err
        }

        tagsValid, err := tc.isTagsValid(item.Tags)
        if err != nil {
                return fmt.Errorf("isTagsValid error %s", err)
        }

        if tagsValid {
                err = json.Unmarshal(item.Data, &in)
                return err
        }

        return tc.rebuild(mkey, ttl, in, rebuildCb)
}
```

Итак, на вход фцнкция принимаем mkey, ttl и пустой интерфейс — напомним, что это универсальная функция, она должна работать с любыми данными. Сначала мы немножко себя обезопасим — проверим через reflect типы данных, которые мне прислали. В данном случае хотим проверить, что прислали действительно указатель на какую-то структуру, то есть если записать результат, он точно наверху появится. Дальше вызываем checklock. Спустимся на уровень ниже. Что такое checklock, зачем он?

```
func (tc *TCache) checkLock(mkey string) error {
        for i := 0; i < 4; i++ {
                _, err := tc.Get("lock_" + mkey)
                if err == memcache.ErrCacheMiss {
                        return nil
                }
                if err != nil {
                        return err
                }
                time.Sleep(10 * time.Millisecond)
        }
        return nil
}
```

Она проверяет, есть ли уже lock на этот кеш, не перестраивает ли его кто-то в данный момент. В ней мы пытаемся несколько раз сходить в memcache по ключу lock + mkey, и если вдруг видим, что записи нет, то это значит, что никто её не перестраивает, я можно идти дальше, В противном случае ждем, пока кто-то перестроит. Стоит обратить внимание, что это учебный пример. У вас может быть другая логика, вы можете спать дольше или вести себя по-разному зависимости от кэша. Теперь возвращаемся в функцию TGet. Когда дождались, получаем уже сами записи из кэша, уже по mkey (без префикса *lock_*).

Если столкнулись с ситуацией, когда записи в кэше еще нет и нужно теперь перестроить кэш, вызываем функцию rebuilt, куда передаем фактически все те же данные — mkey, время жизни, переменную, куда нужно записать, и callback, который вернет настоящие данные. В rebuild углубимся чуть позже. Если же значение в кэше у нас нашлось. То его нужно распаковать из json. Пришло время посмотреть, как выглядит структура CacheItem.

```
type CacheItem struct {
        Data json.RawMessage
        Tags map[string]int
}
```

Обратите внимание, в Data используем параметр json.RawMessage. Это значит, что, даже если там будут дальше фигурные скобки, они не будут распаковываны дальше во что-то другое, а сохранятся прямо как есть, как слайс байт. Это понадобится чуть дальше. Возвращаемся назад. Распаковали, теперь у нас есть какие-то данные. Мы еще не знаем, какова их структура. Поэтому не можем распаковать их до конца. Но уже можем распаковать теги. Теперь нужно пойти и проверить, валидны ли теги.

Как устроен внутри isTagsValid?

```
func (tc *TCache) isTagsValid(itemTags map[string]int) (bool, error) {
        tags := make([]string, 0, len(itemTags))
        for tagKey := range itemTags {
                tags = append(tags, tagKey)
        }

        curr, err := tc.GetMulti(tags)
        if err != nil {
                return false, err
        }
        currentTagsMap := make(map[string]int, len(curr))
        for tagKey, tagItem := range curr {
                i, err := strconv.Atoi(string(tagItem.Value))
                if err != nil {
                        return false, err
                }
                currentTagsMap[tagKey] = i
        }
        return reflect.DeepEqual(itemTags, currentTagsMap), nil
}
```

Для начала конструируем слайс строк. Потом используем метод GetMulti из memcache-клиента. Он получает сразу много записей. По ним строим мапу текущих записей. Конвертируем вернувшиеся memcache item'ы: проходимся по ним, преобразуем слайс байт в int и кладем в мапу. Теперь у нас есть две мапы — одна мапа, которая лежала в кэше, и вторая мапа с текущими значениями, актуальными. Используя функцию DeepEqual из пакета reflect сравниваем, действительно ли обе эти мапы полностью равны. Если они равны, значит кэш валиден. Если не равны, значит кэш не валиден, его нужно перестроить.

Посмотрели, как работает проверка валидности кешей, теперь опять возвращаемеся обратно. Допустим, кэш валиден. Это значит, что надо распаковать данные этого кэша в реальные данные — в ту структуру, которую хотим видеть на выходе. Напомним, что мы не могли распаковать данные раньше, потому что не знали, что там и надо ли это распаковывать, а теперь знаем. Передаем данные и переменную, куда надо записать результат в json.Unmarshal, который, за счет того, что он внутри работает через reflect динамически, пройдется по данным, структуры которых мы раньше не знали, и сможет их записать туда, куда нам нужно. То есть в даном случае мы можем работать абсолютно с любыми данными, сохраненными в кэше, используя всего лишь несколько этапов распаковки.

Теперь, как поступать, если кэш не валиден? Его надо перестроить. Как раз этим занимается функция rebuild.

```
func (tc *TCache) rebuild(
        mkey string,
        ttl int32,
        in interface{},
```

```
        rebuildCb RebuildFunc,
) error {
        tc.lockRebuild(mkey)
        defer tc.unlockRebuild(mkey)

        result, tags, err := rebuildCb()

        // ожидаем и возвращаем одинаковые типы
        if reflect.TypeOf(result) != reflect.TypeOf(in) {
                return fmt.Errorf(
                        "data type mismatch, expected %s, got %s", reflect.TypeOf(in),
                        reflect.TypeOf(result),
                )
        }

        currTags, err := tc.getCurrentItemTags(tags, ttl)
        if err != nil {
                return err
        }

        cacheData := CacheItemStore{result, currTags}
        rawData, err := json.Marshal(cacheData)
        if err != nil {
                return err
        }

        err = tc.Set(&memcache.Item{
                Key:        mkey,
                Value:      rawData,
                Expiration: int32(ttl),
        })

        inVal := reflect.ValueOf(in)
        resultVal := reflect.ValueOf(result)
        rv := reflect.Indirect(inVal)
        rvpresult := reflect.Indirect(resultVal)
        rv.Set(rvpresult) // *in = *result

        return nil
}
```

В функцию передаем ключ, время жизни, переменную, куда хотим записать результат, и callback, который вернет реальные данные. Теперь, что происходит в самой функции. Для начала залочим кэш на перестроение с помощью функции lockRebuild.

```
func (tc *TCache) lockRebuild(mkey string) (bool, error) {
        // пытаемся взять лок на перестроение кеша
        // чтобы все не ломанулись его перестраивать
        // параметры надо тюнить
        lockKey := "lock_" + mkey
        lockAccuired := false
        for i := 0; i < 4; i++ {
                // add добавляет запись если её ещё нету
                err := tc.Add(&memcache.Item{
                        Key:        lockKey,
                        Value:      []byte("1"),
                        Expiration: int32(3),
                })

                if err == memcache.ErrNotStored {
                        fmt.Println("get lock try", i)
                        time.Sleep(time.Millisecond * 10)
```

```
                        continue
                } else if err != nil {
                        return false, err
                }
                lockAccuired = true
                break
        }
        if !lockAccuired {
                return false, fmt.Errorf("Can't get lock")
        }
        return true, nil
}
```

В ней в цикле несколько раз будем пробовать делать add на запись. Почему add, а не set? Add добавляет запись, только если её там нет. Это значит, что, если в данном случае сделаем add и запись не получится добавить — ErrNotStored, то значит кто-то другой перестраивает эту запись, и нам следует немножко подождать. Ставим timeout на 3 секунды. Еще раз обратите внимание, что это учебный пример и все цифры тут учебные. Если мы дождались и смогли взять lock, то, значит, смогли успешно добавить запись в memcache. Из функции возвращаемся, вернув true, если успешно взяли lock на перестроение. Сразу же в defer вызываем unlockRebuild, который просто удалит запись из мемкэша, и теперь в rebuild'е можно продолжать работать. Вызываем callback, который я тащили с самого начала. Он возвращает результат — пустой интерфейс. Теперь проделаем немножко проверок. Хотим убедиться, что то, что запрашиваем извне, и тот результат, который вернул callback, имеют одинаковые типы. Согласитесь, будет очень странно, если вы будете присваивать в разные типы данных. Проверяем через reflect.TypeOf. Он возвращает тип данных, которые там находятся. rebuildCb возвращает теги в виде слайса строк, мы же хотим получить их значения из memcache.

```
func (tc *TCache) getCurrentItemTags(tags []string, ttl int32) (map[string]int, error) {
        currTags, err := tc.GetMulti(tags)
        if err != nil {
                return nil, err
        }
        resultTags := make(map[string]int, len(tags))
        now := int(time.Now().Unix())
        nowBytes := []byte(fmt.Sprint(now))

        for _, tagKey := range tags {
                tagItem, tagExist := currTags[tagKey]
                if !tagExist {
                        err := tc.Set(&memcache.Item{
                                Key:        tagKey,
                                Value:      nowBytes,
                                Expiration: int32(ttl),
                        })
                        if err != nil {
                                return nil, err
                        }
                        resultTags[tagKey] = now
                } else {
                        i, err := strconv.Atoi(string(tagItem.Value))
                        if err != nil {
                                return nil, err
                        }
                        resultTags[tagKey] = i
                }
        }
        return resultTags, nil
}
```

Деляем это опять с помощью отдельной функции, в ней этого мы по слайсу тегов делаем GetMulti, получаем мапу и смотрим, если такая запись в memcache есть, то берем её значение из memcache, если же записи нет, то я создаем её, используя текущее время. Теперь снова возвращаемся на уровень выше —

в нашем распоряжении map из текущих значений, которые лежат в memcache по тегам. И мы хотим их записать в структуру и сохранить.

```
type CacheItemStore struct {
        Data interface{}
        Tags map[string]int
}
```

Вот так выглядит структура, которую будем сериализовать в JSON. В качестве data там пустой интерфейс, это значит, туда можно присвоить любые данные, и когда мы будем их сериализовать, все отработает через reflect или сгенеренные маршалеры. Замаршаливаем данные, добавляем в структуру и устанавливаем её в мемкеш через set. Теперь каким-то образом нужно присвоить result. Идея написать in = result не сработает, потому что внутри интерфейс, написать *in = *result тоже не получится, все присвоится немножко не так, как вы ожидаете. Поэтому дальше идет хитрое присвоение. Продублируем его для наглядности.

```
inVal := reflect.ValueOf(in)
resultVal := reflect.ValueOf(result)
rv := reflect.Indirect(inVal)
rvpresult := reflect.Indirect(resultVal)
rv.Set(rvpresult) // *in = *result
```

Поскольку это интерфейсы, то можем достучаться до значения, которое лежит в глубине. Однако мы не знаем, какое именно значение в реальности будет, поэтому сначала получим ValueOf от той переменной, которую хотим записать и той, куда хотим записать. Теперь, используя indirect, то есть direct получает Value, куда указывает предыдущая Value — это уже прямо магия — получаем оба значения, и теперь, наконец, можно сделать set у этого значения. То есть теперь, наконец, достучались до самых-самых-самых переменных, которые лежат за пустым интерфейсом, и можем присвоить их значение, то есть записать финальный результат, который прокинется на самый верх.

Запустите самостоятельно пример и попробуйте до конца осознать, что тут все-таки происходит. Пример, конечно, сложный, но зато тут демонстрируется довольно много возможностей самого языка. Стоит оговорить еще раз, что это учебный пример, однако подход, использованный в нем рабочий, поэтому, возможно, вам стоит посмотреть на инвалидацию кэша, немножко доработать ее под себя. Например, если кэш не валиден, то возвращать старые данные. Или, например, вы захотите перестраивать кэш, просто запуская его, запуская функцию в отдельной горутине, а те данные, которые есть, возвращать уже наверх, даже несмотря на то, что они немножко устарели, вы будете просто знать, что они там сейчас перестроятся.

Redis

В этом разделе поговорим про redis. Изначально redis был мемкэшем на стероидах, однако сейчас это key value хранилище общего назначения, которое поддерживает просто огромное количество команд буквально на все случаи жизни. Мы будем рассматривать его уже на знакомом примере с менеджером сессии и будем хранить эти сессии в redis. Сразу стоит отметить, что это будет учебный пример. Если вы хотите использовать redis для хранения сессии, лучше используйте готовый протестированный модуль. В нем будет гораздо больше возможностей.

Redis, конечно, не поставляется с Go, это внешняя библиотека, устанавливается она, как и все остальное через go get. В коде перед началом использования нужно установить соединение к redis. Делается это через DialURL. URL задается через флаги по умолчанию. Далее мы создаем менеджер сессии и в него передаем соединение до redis.

```
package main

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

        "github.com/garyburd/redigo/redis"
```

```
)

var (
        redisAddr = flag.String("addr", "redis://user:@localhost:6379/0",
                                                        "redis addr")

        sessManager *SessionManager

        users = map[string]int{
                "rvasily":        "test",
                "romanov.vasily": "100500",
        }
)

func main() {
        flag.Parse()

        var err error
        redisConn, err := redis.DialURL(*redisAddr)
        if err != nil {
                log.Fatalf("cant connect to redis")
        }

        sessManager = NewSessionManager(redisConn)

        http.HandleFunc("/", innerPage)
        http.HandleFunc("/login", loginPage)
        http.HandleFunc("/logout", logoutPage)

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

Теперь посмотрим, какими командами все это выполняется. В менеджере сессии нем ничего нет, кроме соединения до самого redis.

```
func NewSessionManager(conn redis.Conn) *SessionManager {
        return &SessionManager{
                redisConn: conn,
        }
}
```

Наш менеджер сессии очень простой, он не поддерживает никакие переменные, которые хранятся внутри сессии, не поддерживает ничего, просто сохранение логина и юзер-агента. При попытке создать сессию, создаем какой-то случайный ключ, сериализуем данные в json. Затем выполняем команду set внутри функции do у соединения, которая отправляет все команды и значения в сам redis. Функций do возвращает какие-то данные и ошибку. Поскольку данные, которые возвращает сессия, не понятно, в каком они формате, мы приводим их Go-типам. В redis есть набор утилитарных функций как раз для конвертации данных. Собственно, это всё что нужно, чтобы установить значение в redis. Очень похоже на мемкэш.

```
func (sm *SessionManager) Create(in *Session) (*SessionID, error) {
        id := SessionID{RandStringRunes(sessKeyLen)}
        dataSerialized, _ := json.Marshal(in)
        mkey := "sessions:" + id.ID
        result, err := redis.String(sm.redisConn.Do("SET", mkey,
                                        dataSerialized, "EX", 86400))
        if err != nil {
                return nil, err
        }
        if result != "OK" {
                return nil, fmt.Errorf("result not OK")
        }
```

```
        return &id, nil
}
```

Теперь, как выглядить проверка.

```
func (sm *SessionManager) Check(in *SessionID) *Session {
        mkey := "sessions:" + in.ID
        data, err := redis.Bytes(sm.redisConn.Do("GET", mkey))
        if err != nil {
                log.Println("cant get data:", err)
                return nil
        }
        sess := &Session{}
        err = json.Unmarshal(data, sess)
        if err != nil {
                log.Println("cant unpack session data:", err)
                return nil
        }
        return sess
}
```

Тут все почти то же самое. Только выполняем команду не set, а get на ключ и получаем данные. Обратите внимание, команда set возвращает, get же возвращает slice byte, поэтому для преобразования используем хелпер bytes, а далее распаковываем в нем json.

Наконец, команда delete, которая удаляет значения.

```
func (sm *SessionManager) Delete(in *SessionID) {
        mkey := "sessions:" + in.ID
        _, err := redis.Int(sm.redisConn.Do("DEL", mkey))
        if err != nil {
                log.Println("redis error:", err)
        }
}
```

Точно так же вызываем метод Do у соединения, и парсим его. Ответ распаковываем при помощи Int-хелпера (команда delete возвращает число записей, удаленных по данному ключу), но используем дальше только ошибку, чтобы проверить, все ли прошло успешно.

Напоследок немного про общую логику осталось упомянуть. При работе непосредственно с пользователем SessionManager будем SessionID получать и сверять по Cookie и отправлять пользователя на форум авторизации, либо же показывать ему какой-то текст. При создание сессии тоже все просто — создаем структуру, кде указываем свои данные и юзер-агента.

Прочие системы

Message broker - RabbitMQ

В этом разделе мы научимся работать с RabbitMQ. Rabbit — платформа, реализующая систему обмена сообщениями между компонентами программной системы с использованием паттерна publisher-subscriber. Её полезно использовать, когда вы хотите отправлять разного рода тяжелые операции из запроса пользователя в очередь на выполнение оффлайн-воркером и не заставлять пользователя ждать, пока завершится операция. Например, если есть сервис, конвертирующий видео, он не станет очень долго крутить индикатор загрузки. Он загрузит видео, положит его на диск и скажет пользователю: ваше видео обрабатывается, подождите немножко. В это время видео попадет в очередь и будет ждать, пока другой процесс, скорее всего работающий на другом сервере, начнет конвертировать его.

Мы в качестве примера рассмотрим операцию не такую тяжелую, как конвертация видео — мы будем создавать превью для картинок. Рассмотрим код, из которого должно стать понятно, каким образом все происходит.

```
var (
        rabbitAddr = flag.String("addr",
```

```
            "amqp://guest:guest@192.168.99.100:32778/", "rabbit addr")

        rabbitConn *amqp.Connection

        rabbitChan *amqp.Channel
)

func main() {
        flag.Parse()
        var err error
        rabbitConn, err = amqp.Dial(*rabbitAddr)
        panicOnError("cant connect to rabbit", err)

        rabbitChan, err = rabbitConn.Channel()
        panicOnError("cant open chan", err)
        defer rabbitChan.Close()

        q, err := rabbitChan.QueueDeclare(
                ImageResizeQueueName, // name
                true,                 // durable
                false,                // delete when unused
                false,                // exclusive
                false,                // no-wait
                nil,                  // arguments
        )
        panicOnError("cant init queue", err)

        fmt.Printf("queue %s have %d msg and %d consumers\n",
                q.Name, q.Messages, q.Consumers)

        http.HandleFunc("/", mainPage)
        http.HandleFunc("/upload", uploadPage)

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

Для начала, для начала мы выполняем подключение к Rabbit. Обратите внимание, что соединение к Rabbit не локальная переменная, а глобальная. Это надо для того, чтобы им можно было воспользоваться из других функций. После создания подключения надо получить канал, через который будет проходить общение с Rabbit. Определим очередь сообщений — зададим имя, количество сообщений и консьюмеров. Сейчас в ней 0 сообщений и один консьюмер. Консьюмер — это оффлайн-воркер, наш гошный сервис, который принимает задачи из очереди. Затем стартуем сервер.

Каким образом мы взаимодействуем с пользователем и как класть сообщение в очередь?

```
func uploadPage(w http.ResponseWriter, r *http.Request) {
        uploadData, handler, err := r.FormFile("my_file")
        if err != nil {
                fmt.Println(err)
                return
        }
        defer uploadData.Close()

        fmt.Fprintf(w, "handler.Filename %v\n", handler.Filename)
        fmt.Fprintf(w, "handler.Header %#v\n", handler.Header)

        tmpName := RandStringRunes(32)

        tmpFile := "./images/" + tmpName + ".jpg"
        newFile, err := os.Create(tmpFile)
        if err != nil {
```

```
                http.Error(w, "cant open file: "+err.Error(),
                                        http.StatusInternalServerError)
                return
        }

        hasher := md5.New()
        writtenBytes, err := io.Copy(newFile,
                        io.TeeReader(uploadData, hasher))
        if err != nil {
                http.Error(w, "cant save file: "+err.Error(),
                                        http.StatusInternalServerError)
                return
        }
        newFile.Sync()
        newFile.Close()

        md5Sum := hex.EncodeToString(hasher.Sum(nil))

        realFile := "./images/" + md5Sum + ".jpg"
        err = os.Rename(tmpFile, realFile)
        if err != nil {
                http.Error(w, "cant raname file: "+err.Error(),
                                        http.StatusInternalServerError)
                return
        }

        data, _ := json.Marshal(ImgResizeTask{handler.Filename, md5Sum})

        fmt.Println("put task ", string(data))

        err = rabbitChan.Publish(
                "",                     // exchange
                ImageResizeQueueName, // routing key
                false,                  // mandatory
                false,
                amqp.Publishing{
                        DeliveryMode: amqp.Persistent,
                        ContentType:  "text/plain",
                        Body:         data,
                })
        panicOnError("cant publish task", err)

        fmt.Fprintf(w, "Upload %d bytes successful\n", writtenBytes)
}
```

Для начала мы вычитаем файл из ответа — для этого у нас есть функция FormFile. Напечатаем какую-то информацию на экран пользователю . Поскольку использовать имя файла, переданное пользователем, крайне не безопасно, сгенерируем случайное имя. А затем провернем трюк — одновременно в один проход, построим и md5 от файла, и запишем файл на диск. Для этого есть TeeReader, куда можно передать еще один ридер. Переименовываем картинку по реальному md5. Теперь уже можно создать задачу: запаковываем все, что нужно воркеру знать о задаче. В реальной жизни у вас, скорее всего, в эту задачу войдет еще и адрес, чтобы ваш фреймворк знал, на каком хранилище она сохранена. Выводим сконструированное сообщение пользователю и публикуем его в Rabbit. Обратите внимание: rabbitChan — это канал общения с Rabbit, не гошный канал для общения с горутинами.

Обратите внимание: в пользовательском запросе мы просто кладем файл на диск, но не пережимаем его. Пользователь получит сообщение и обработка запроса завершится, сама же работа будет происходить отдельно от запрроса — в обработчике. Рассмотрим, как реализовать получение и обработку сообщения из очереди.

```
var (
        rabbitAddr = flag.String("addr",
```

```
                "amqp://guest:guest@192.168.99.100:32778/", "rabbit addr")

        rabbitConn *amqp.Connection

        rabbitChan *amqp.Channel
)

func main() {
        flag.Parse()
        var err error
        rabbitConn, err = amqp.Dial(*rabbitAddr)
        panicOnError("cant connect to rabbit", err)

        rabbitChan, err = rabbitConn.Channel()
        panicOnError("cant open chan", err)
        defer rabbitChan.Close()

        _, err = rabbitChan.QueueDeclare(
                ImageResizeQueueName, // name
                true,                 // durable
                false,                // delete when unused
                false,                // exclusive
                false,                // no-wait
                nil,                  // arguments
        )
        panicOnError("cant init queue", err)

        err = rabbitChan.Qos(
                1,     // prefetch count
                0,     // prefetch size
                false, // global
        )
        panicOnError("cant set QoS", err)

        tasks, err := rabbitChan.Consume(
                ImageResizeQueueName, // queue
                "",                   // consumer
                false,                // auto-ack
                false,                // exclusive
                false,                // no-local
                false,                // no-wait
                nil,                  // args
        )
        panicOnError("cant register consumer", err)

        wg := &sync.WaitGroup{}
        wg.Add(1)

        go ResizeWorker(tasks)

        fmt.Println("worker started")
        wg.Wait()
}
```

Начинаем мы, как всегда с подключения к Rabbit — мы не можем им пользоваться, не подключившись. Опять получаем канал, определяем очередь, устанавливаем, какое количество сообщений мы хотим подтягивать из очереди в свой буфер, для того чтобы быстро их обрабатывать. Теперь получаем канал уже, из которого мы будем читать сообщения, то есть в данном случае tasks — это будет уже гошный канал. И по нему мы можем итерироваться, используя for range или select. В отдельной горутине вызываем воркера, который и будет работать тасками. При этом обработчик легко масштабировать, создавая нужное количество горутин в цикле.

Как устроен воркер?

```
func ResizeWorker(tasks <-chan amqp.Delivery) {
        for taskItem := range tasks {
                fmt.Printf("incoming task %+v\n", taskItem)

                task := &ImgResizeTask{}
                err := json.Unmarshal(taskItem.Body, task)
                if err != nil {
                        fmt.Println("cant unpack json", err)
                        taskItem.Ack(false)
                        continue
                }

                originalPath := fmt.Sprintf("./images/%s.jpg", task.MD5)
                for _, size := range sizes {
                        resizedPath := fmt.Sprintf("./images/%s_%d.jpg",
                                                        task.MD5, size)
                        err := ResizeImage(originalPath, resizedPath, size)
                        if err != nil {
                                fmt.Println("resize failed", err)
                        }
                        time.Sleep(3 * time.Second)
                }

                taskItem.Ack(false)
        }
}
```

В воркере мы просто итерируемся через for range по каналу. Получаем очередной taskItem, распаковываем его и, если распаковка прошла успешно, и для нескольких размеров картинок вызываем сжимающую функцию ResizeImage, куда передаем оригинальный путь, новый путь для картинки и размер, до которого надо ужать.

То есть фактически задача сводится буквально к тому, чтобы просто читать из канала и выполнять работу. То есть если бы это было на одном сервере, можно было бы обойтись каналами внутри Go. Картинки пережимались бы в фоне от пользовательских запросов. Но в реальность, если у вас есть потребность пережимать картинки через очередь, скорее всего, этих картинок у вас много, серверов у вас тоже много. И вы не можете себе позволить такую роскошь как грузить сервера, которые обрабатывают пользовательские запросы, еще и сжиманием картинок. Поэтому это выносится на отдельный сервер.

Document store - MongoDB

Напоследок поговорим про использование MongoDB в Go. MongoDB относится к классу Document storage, то есть это уже не SQL-система, скорее no-SQL. Mongo примечателен тем, что в базе данных, которая там есть, нет схемы. Все данные хранятся в виде BSON — это такой бинарный JSON. То есть туда можно добавлять любые поля, удалять любые поля, при этом по полям, которые там есть,можно строить индекс. MongoDB удобен в применении, если у вас очень много различных атрибутов, которые абсолютно новые для разных записей.

Рассмотрим на примере, как работать с Mongo. Mongo — это, как я уже говорил, не SQL. То есть драйвер может быть совсем не похож, а может быть и похож на то, что вы видели.

```
package main

import (
        "fmt"
        "html/template"
        "net/http"
        "strconv"
```

```
        mgo "gopkg.in/mgo.v2"
        "gopkg.in/mgo.v2/bson"

        "github.com/gorilla/mux"
)

type Item struct {
        Id          bson.ObjectId `json:"id" bson:"_id"`
        Title       string        `json:"title" bson:"title"`
        Description string        `json:"description" bson:"description"`
        Updated     string        `json:"updated" bson:"updated"`
}

func main() {
        sess, err := mgo.Dial("mongodb://localhost")
        __err_panic(err)

        // если коллекции не будет, то она создастся автоматически
        collection := sess.DB("coursera").C("items")

        // для монги нет такого красивого дампа SQL, поэтому вставляем демо-запись
        if n, _ := collection.Count(); n == 0 {
                collection.Insert(&Item{
                        bson.NewObjectId(),
                        "mongodb",
                        "Рассказать про монгу",
                        "",
                })
                collection.Insert(&Item{
                        bson.NewObjectId(),
                        "redis",
                        "Рассказать про redis",
                        "rvasily",
                })
        }

        handlers := &Handler{
                Items: collection,
                Tmpl:  template.Must(template.ParseGlob("./templates/*")),
        }

        // в целям упрощения примера пропущена авторизация и csrf
        r := mux.NewRouter()
        r.HandleFunc("/", handlers.List).Methods("GET")
        r.HandleFunc("/items", handlers.List).Methods("GET")
        r.HandleFunc("/items/new", handlers.AddForm).Methods("GET")
        r.HandleFunc("/items/new", handlers.Add).Methods("POST")
        r.HandleFunc("/items/{id}", handlers.Edit).Methods("GET")
        r.HandleFunc("/items/{id}", handlers.Update).Methods("POST")
        r.HandleFunc("/items/{id}", handlers.Delete).Methods("DELETE")

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

// не используйте такой код в прошакшене
// ошибка должна всегда явно обрабатываться
func __err_panic(err error) {
        if err != nil {
                panic(err)
        }
```

```
}
```

Для создаем сессию, и коннектимся к хосту. Дальше коннектимся к базе. Причем, если базы нету, то она автоматически создастся. Создаем в базе коллекцию айтемов. В Mongo нету красивого дампа, как в SQL. Поэтому изначально проверяем, сколько записей в коллекции, если их нет, создаем какие-то демоданные при помощи insert. У нас есть уже знакомая структура — item, однако, objectID тут уже не не int, который автоинкрементится, а bson objectID. И в bson'е прописано спецполе, что подчеркивание id там это как раз-таки означает id'шник. Автоинкремента из коробки в монге нет, но есть сторонний модуль, его доставить в сучае необходимости. В таблице создаем две записи с использованием нашей структуры. Теперь в handler передаем конкретную коллекцию (не сессию).

Рассмотрим все основные операции. Как выбрать из таблицы все записи?

```
func (h *Handler) List(w http.ResponseWriter, r *http.Request) {

        items := []*Item{}

        // bson.M{} - это типа условия для поиска
        err := h.Items.Find(bson.M{}).All(&items)
        __err_panic(err)

        err = h.Tmpl.ExecuteTemplate(w, "index.html", struct {
                Items []*Item
        }{
                Items: items,
        })
        if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
}
```

Для этого пользуемся методом find, передавая туда пустой bson с пустым условием для поиска. Все, что найдется, записываем это в слайс item'ов. Дальше уже знакомые действия: экспандим шаблон, передаем туда item'ы.

Как происходит вставка?

```
func (h *Handler) Add(w http.ResponseWriter, r *http.Request) {

        newItem := bson.M{
                "_id":         bson.NewObjectId(),
                "title":       r.FormValue("title"),
                "description": r.FormValue("description"),
                "some_filed":  123,
        }

        err := h.Items.Insert(newItem)
        __err_panic(err)

        fmt.Println("Insert - LastInsertId:", newItem["id"])

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

Вставку можно сделать, создав новую структуру Item и вставив в таблицу. В этом случае в автоинкрементное поле, в отличие от SQL, нельзя передать пустую строку. Или можно сделать вставку при помощи bson, как в примере. Обратите внимание, что в этом случае поля уже должны называться так, как они мапятся на структуру. Кроме того, так мы не привязаны к жесткой структуре, и можем добавлять любые поля. Естественно, выбирать записи можно тоже сразу в bson, а не в структуру. Это удобно, если у вас очень много динамики, но порой бывает тяжело поддерживать, когда схемы нет, и все разлезается.

Update опять состоит из двух частей — редактирование и собственно внесение изменений.

```
func (h *Handler) Edit(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        if !bson.IsObjectIdHex(vars["id"]) {
                http.Error(w, "bad id", 500)
                return
        }
        id := bson.ObjectIdHex(vars["id"])

        post := &Item{}
        err := h.Items.Find(bson.M{"_id": id}).One(&post)

        err = h.Tmpl.ExecuteTemplate(w, "edit.html", post)
        if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
}
```

Здесь идейно происходит все то же самое, что и раньше. Основное, что нужно знать: если пытаться получить objectIDHex от какой-то неправильной строки, то будет паника. Поэтому вначале надо проверить, действительно ли айдишник валидный objectIDHex, и только после этого я создать нужную структуру и делать find. Для поиска указываем поле — id — и значение. Выбираем единичную запись, кладем ее в пост и передаем дальше в шаблон для экспанда. В Update собственно, все тоже просто.

```
func (h *Handler) Update(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        if !bson.IsObjectIdHex(vars["id"]) {
                http.Error(w, "bad id", 500)
                return
        }
        id := bson.ObjectIdHex(vars["id"])

        post := &Item{}
        err := h.Items.Find(bson.M{"_id": id}).One(&post)

        post.Title = r.FormValue("title")
        post.Description = r.FormValue("description")
        post.Updated = "rvasily"

        // err = h.Items.Update(bson.M{"_id": id}, &post)
        err = h.Items.Update(
                bson.M{"_id": id},
                bson.M{
                        "title":       r.FormValue("title"),
                        "description": r.FormValue("description"),
                        "updated":     "rvasily",
                        "newField":    123,
                })
        affected := 1
        if err == mgo.ErrNotFound {
                affected = 0
        } else if err != nil {
                __err_panic(err)
        }

        fmt.Println("Update - RowsAffected", affected)

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

Так же как insert апдейт можно сделать, передав структуру либо же абсолютно произвольный JSON. Единственное, что нужно быть аккуратными с названиями полей — регистр имеет значение.

Осталось посмотреть, как устроено удаление.

```
func (h *Handler) Delete(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        if !bson.IsObjectIdHex(vars["id"]) {
                http.Error(w, "bad id", 500)
                return
        }
        id := bson.ObjectIdHex(vars["id"])

        err := h.Items.Remove(bson.M{"_id": id})
        affected := 1
        if err == mgo.ErrNotFound {
                affected = 0
        } else if err != nil {
                __err_panic(err)
        }

        w.Header().Set("Content-type", "application/json")
        resp := `{"affected": ` + strconv.Itoa(int(affected)) + `}`
        w.Write([]byte(resp))
}
```

Опять таки проверяем, действительно ли передадим objectIDHex, и вызываем метод Remove, куда передаем опять-таки bson с номером поля. Вот все. Ничего сложного.