به نام خدا

دانشگاه تهران

پردیس دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

مبانی رایانش توزیع‌شده

# گزارش نهایی تمرین کامپیوتری شماره 2 (Concurrency in Go)

نام استاد : دکتر شجاعی


نام دانشجویان :

علی مهرانی - 810198542

علی رنجبری - 810198570

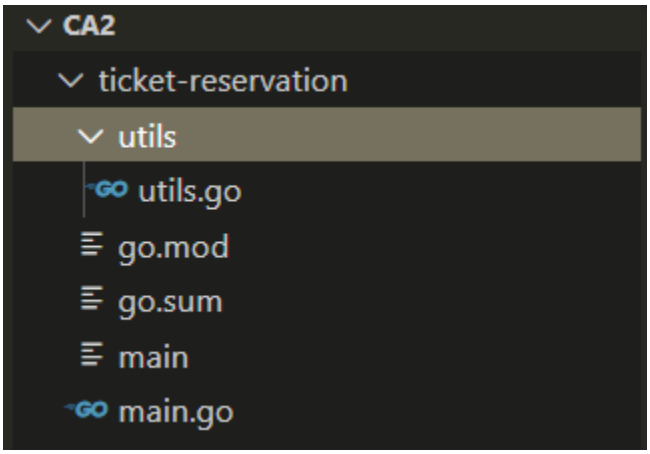
امیررضا قهرمانی - 810198463

# تقسیمات کار پروژه

- **علی مهرانی**
  - بخش‌های 1 و 2 و 3 و 4
  - گزارش بخش‌های مذکور
- **علی رنجبری**
  - بخش‌های 4 و 7
  - گزارش بخش‌های مذکور
- **امیررضا قهرمانی**
  - بخش‌های 5 و 6
  - گزارش بخش‌های مذکور

# ساختار نهایی کد پروژه

شکل زیر ساختار نهایی کد پروژه را نشان می‌دهد.




شکل 1- ساختار نهایی کد پروژه

در ادامه توضیحات مربوط به هرکدام از بخش‌های پروژه آمده است.

# توضیحات بخش‌های پروژه

## 1. راه‌اندازی پروژه جدید

با توجه موارد ذکرشده در این بخش از پروژه، فایل main.go را ساخته و ساختارهای داده لازم را برای Ticket و Event تعریف می‌کنیم.
فایل main.go را ایجاد کرده و ساختارهای داده ذکر شده را در آن ایجاد می‌کنیم.
شکل زیر کدهای ایجاد شده در این بخش را نمایش می‌دهد. (فیلد mutx مربوط به بخش 3 می‌باشد)

```
type Event struct {
    ID                  string
    Name                string
    Date                time.Time
    TotalTickets        int
    AvailableTickets    int
    mutx                sync.Mutex
}

type Ticket struct {
    ID      string
    EventID string
}
```

شکل 2- ساختار اولیه Ticket و Event

## 2. پیاده‌سازی سرویس رزرو بلیت

مطابق با خواسته این بخش، Struct مربوط به TicketService را برای منطق مدیریت Event و رزرو بلیت‌ها پیاده‌سازی می‌کنیم.
شکل زیر کد مربوط به این بخش را نمایش می‌دهد.

```go
type TicketService struct {
    events sync.Map
}

func NewTicketService() *TicketService {
    return &TicketService{}
}
```

شکل 3- ساختار اولیه TicketService

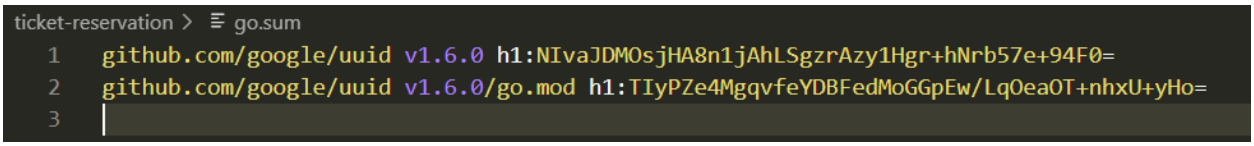
مطابق شکل بالا، Struct با نام TicketService را ایجاد میکنیم که شامل field ای با نام events از نوع Sync.Map می‌باشد. از این نوع داده sync.Map به این دلیل استفاده می‌کنیم که بتوانیم event ها و اطلاعات مربوط به آن‌ها را به صورت Concurrent مدیریت کنیم.

همچنین تابع NewTicketService نیز ایجاد شده که یک constructor برای TicketService می‌باشد و با هربار فراخوانی، یک pointer جدید به یک struct از نوع TicketService ایجاد و برمیگرداند.
در ادامه کدهای مربوط به متدهای CreateEvent و ListEvents و BookEvents را ایجاد و نمایش می‌دهیم (تصاویر آن‌ها در بخش 3 آمده است).

همچنین لازم به ذکر است که برای ایجاد یک ID جدید برای هر event، باید از تابع generateUUID استفاده کنیم، که برای این کار باید از پکیج UUID که ساخت Google است، استفاده کنیم. بنابراین لازم است تا این پکیج را نصب و import کنیم که این کار را با استفاده از دستور زیر انجام می‌دهیم.

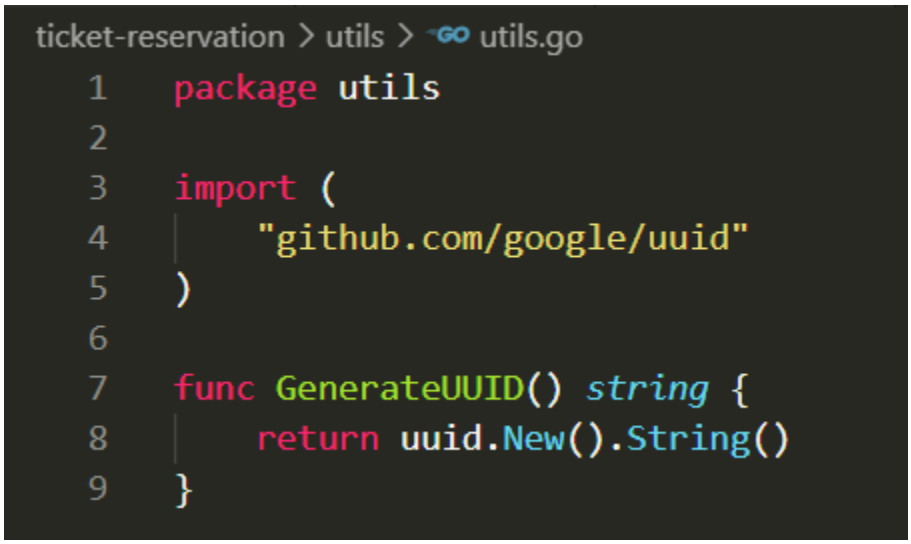
```
go get github.com/google/uuid
```

پس از نصب این پکیج فایلی با عنوان go.sum توسط go ایجاد می‌شود که برای dependancy management می‌باشد و اطلاعات مربوط به dependancy های پروژه در آن قرار می‌گیرد.
شکل زیر محتویات این فایل را پس از نصب پکیج ذکر شده نمایش می‌دهد.




شکل 4- فایل go.sum

در ادامه برای استفاده از این پکیج، یک پکیج با نام utils در پروژه خود تعریف میکنیم و تابع GenerateUUID را در آن تعریف می‌کنیم.
شکل زیر ساختار فایل utils.go را نمایش می‌دهد.




شکل 5- ساختار فایل utils.go

مطابق شکل بالا، تابع ذکر شده یک uuid جدید باز نوع string برمی‌گرداند.

## 3. پیاده‌سازی Concurrency control

در ادامه قسمت Concurrency control را پیاده‌سازی می‌کنیم.
ابتدا در کد اولیه نقاط بحرانی یا critical sections را که دسترسی همزمان ممکن است منجر به Race condition یا Data Inconsistency شود را مشخص می‌کنیم.

شکل زیر تابع پیاده‌سازی شده BookTickets را نمایش می‌دهد که نقاط بحرانی در آن مشخص شده‌اند.

```go
func (ts *TicketService) BookTickets(eventID string, numTickets int) ([]string, error) {
    event, ok := ts.events.Load(eventID)
    if !ok {
        return nil, fmt.Errorf("event not found")
    }
    ev := event.(*Event)

    ev.mutx.Lock()
    defer ev.mutx.Unlock()

    if ev.AvailableTickets < numTickets { //critical section
        return nil, fmt.Errorf("not enough tickets available")
    }

    var ticketIDs []string
    for i := 0; i < numTickets; i++ {
        ticketID := utils.GenerateUUID()
        ticketIDs = append(ticketIDs, ticketID)
        //store the ticket in a seperate data structure if needed
    }

    ev.AvailableTickets -= numTickets
    ts.events.Store(eventID, ev)

    return ticketIDs, nil
}
```

شکل 6- تابع BookTickets

همانگونه که در شکل بالا نیز مشخص است، نقاط بحرانی به شرح زیر می‌باشند:

**مقایسه اولیه تعداد بلیت‌های درخواستی و تعداد بلیت‌های قابل اخذ:** درهنگام مقایسه این مقادیر، به صورت concurrent، مقایسه‌های دیگری نیز توسط goroutine های دیگر ممکن است صورت بگیرد که منجر به booking شود. در صورت عدم هماهنگ سازی و synchronization مناسب، امکان بروز race condition وجود دارد.

**فرآیند رزرو بلیت یا Booking:** در حلقه مشخص‌شده، فرآیند رزرو بلیت صورت می‌گیرد که به طور همزمان، تغییر مقدار AvailableTickets در صورت عدم synchronization مناسب، می‌تواند منجر به data inconsistency شود.

**بروزرسانی مقدار بلیت‌های باقی‌مانده:** درقسمت مشخص‌شده، کاهش تعداد بلیت‌ها پس از تکمیل فرآیند رزرو در صورت عدم synchronization مناسب، می‌تواند منجر به data inconsistency شود چون توسط goroutine های متعدد به صورت concurrent بروزرسانی می‌شود.

در ادامه توابع CreateEvent و ListEvents را ایجاد می‌کنیم.

شکل زیر توابع CreateEvent و ListEvents را نمایش می‌دهد.

```go
func (ts *TicketService) CreateEvent(name string, date time.Time, totalTickets int) (*Event, error) {
    event := &Event{
        ID:               utils.GenerateUUID(),
        Name:             name,
        Date:             date,
        TotalTickets:     totalTickets,
        AvailableTickets: totalTickets,
    }

    ts.events.Store(event.ID, event)
    return event, nil
}

func (ts *TicketService) ListEvents() []*Event {
    var events []*Event
    ts.events.Range(func(key, value interface{}) bool {
        event := value.(*Event)
        events = append(events, event)
        return true
    })
    return events
}
```

شکل 7- توابع CreateEvent و ListEvents

در تابع createEvent با دریافت اطلاعات بلیت، یک ساختار داده Event جدید ایجاد می‌شود و در داده ساختار events که از نوع sync.Map است، ذخیره می‌کنیم.
در تابع ListEvents نیز، event های مورد نظر را جمع آوری کرده و برمی‌گردانیم.

مطابق شکل 2 در داده ساختار Event یک فیلد به نام mutx اضافه شده که با کمک آن دسترسی به فیلد AvailableTickets در این ساختار synchronize می‌شود.

در تابع bookFunctions از ev.mutx.lock() استفاده کردیم تا قبل از تغییر AvailableTickets یک Lock در اختیار کنیم. سپس از defer ev.mu.Unlock استفاده میکنیم تا بعد از گذر از critical section، قفل را release کنیم و با این کار مطمئن می‌شویم تا فراخوانی‌های موازی BookTickets برای یک event یکسان، دچار race condition یا data inconsistency نشود و فرآیند synchronized باشد.
در ادامه توضیحات بخش 4 آمده است.

## 4. پیاده‌سازی Client Interface

در این بخش برای گرفتن command ها از یک goroutine به اسم commandHandler استفاده میکنیم که به غیر از کامند های help و exit بقیه ی کامند ها یک goroutine دارند که به صورت سریال اجرا میشوند و برای گرفتن پاسخ ها از دو تابع outputManager و errorManager استفاده میکنیم که به صورت موازی هر پاسخی که بیاید را روی ترمینال چاپ میکنیم ، تابع commandHandler به صورت زیر پیاده سازی شده است

```go
func CommandHandler(commands []string, processManager *ProcessManager, ticketService *TicketService, chErr chan error, chOutput chan string) {

    processManager.mutx.Lock()
    processManager.ID++
    processManager.numProcessRunning++
    id := processManager.ID
    processManager.mutx.Unlock()
    defer processManager.wg.Done()

    chOutput <- fmt.Sprintf("Your request with id %d is processing ... \n", id)

    switch commands[0] {
    case "create":
        CreateTicketCommand(id, ticketService, commands, chErr, chOutput)

    case "book":
        BookTicketsCommand(id, ticketService, commands, chErr, chOutput)

    case "list":
        ticketService.PrintListEvents(chOutput)
    }
}
```

شکل 8- تابع commandHandler

بدین صورت interface ما به این شکل میشود :

```
const HELP_MESSAGE = `
Commands are:
    * create new event:
        - create <name: str> <date: 2006-01-02_15:04:05> <numTickets: int>
    * book tickets:
        - book <eventId: str> <numTickets: int>
    * listing events:
        - list
    * help menue:
        - help
    * exiting the program:
        - exit`
```

شکل 9- ساختار interface

## 5. پیاده‌سازی Fairness و Resource management

از آن جایی که goroutine ها خودشان یک backend job هستند ، طبق الگوریتم هایی که در سیستم عامل وجود دارد schedule میشوند و دیگر به starvation برخورد نمیکنند و برای resource management ما از یک structure به نام ProcessManager استفاده کردیم که میتواند برای process ها id تولید کند و یکی دیگر از field ها در این استراکچر ، numProcessRunning نام دارد که میتوانیم تعداد backend job هایی که اجرا میشود را با استفاده از آن ست کنیم و بیشتر از آن اجازه ی اجرا شدن درخواستی را ندهیم ، همچنین یک field دیگر به اسم wg یا waitGroup وجود دارد که برای مثال اگر wg ما n باشد ، صبر میکند تا n تا دستور ما انجام شود و بعد از آن شروع به گرفتن درخواست جدید میکند.

```
type ProcessManager struct {
    ID                int
    numProcessRunning int
    mutx              sync.Mutex
    wg                sync.WaitGroup
}
```

شکل 10- ساختار ProcessManager

## 6. پیاده‌سازی Logging و Error handling

برای logging و Error handling همانگونه که در بخش چهارم توضیح داده شد ، ما از دو تابع outputManager و errorManager استفاده میکنیم که چنل های error و message را میگیرند و هر output ای روی این ها بیاید درجا چاپ میشود و به تمامی تابع های دیگر نیز این دو تابع را پاس داده ایم که اگر output ای داشتند ، این دو تابع ، آن output را چاپ کنند.

این دو تابع به این شکل پیاده سازی شده اند :

```go
func OutputManager(ch <-chan string) {
    for v := range ch {
        fmt.Println(v)
    }
}

func ErrorManager(ch <-chan error) {
    for e := range ch {
        fmt.Printf("**** Error: %s ****\n", e.Error())
    }
}
```

شکل 11- توابع OutputManager و ErrorManager

## 7. پیاده‌سازی Caching

برای پیاده سازی این بخش ما ابتدا یک structure به نام Cache درست کردیم که از آن در یک structure دیگر به نام TicketService استفاده کنیم.

```go
type Cache struct {
    accessCount sync.Map
    events      [NUM_CACHE]*Event
    // mutxs           [NUM_CACHE]sync.Mutex
    accessCountMutx sync.Mutex
}

type TicketService struct {
    cache  Cache
    events sync.Map
}
```

شکل 11- Cache و TicketSerive

حال فیلد های موجود را بررسی میکنیم ، فیلد accessCount در Cache به این صورت است که هربار که کسی بخواهد event ای را book کند ، این accessCount یکی زیاد میشود و تعداد دسترسی به event های مختلف را میشمارد و به تعداد numberOfCache که از قبل تعیین شده است ، اون تعدادی که بیشترین دسترسی را داشته اند را در خود ذخیره میکند که به آنها دسترسی سریع تری داشته باشد، همچنین ما از چندین تابع مثل addAccessCountAndUpdateCache استفاده میکنیم که هر event ای که درخواستی داشته باشد ما با کال کردن این تابع ، میگوییم که accessCount آن را یکی افزایش بده ، و پس از هربار آن تعداد numberOfCache که در خودش ذخیره کرده است را update میکند .
تابع addAccessCountAndUpdateCache به این صورت پیاده سازی شده است :

```go
func (cache *Cache) AddAccessCountAndUpdateCache(eventId string, event *Event) {
    count, ok := cache.accessCount.Load(eventId)
    if !ok {
        cache.accessCount.Store(eventId, 1)

    } else {
        cache.accessCount.Store(eventId, count.(int)+1)
    }

    // update accessCount
    cache.accessCountMutx.Lock()
    defer cache.accessCountMutx.Unlock()
    for i, ev := range cache.events {
        if ev == nil {
            cache.events[i] = event
            break
        }
        if ev == event {
            if i == 0 {
                break
            }
            higherAccess, _ := cache.accessCount.Load(cache.events[i-1].ID)
            lowerAccess, _ := cache.accessCount.Load(event.ID)
            if lowerAccess.(int) > higherAccess.(int) { // swap
                cache.events[i] = cache.events[i-1]
                cache.events[i-1] = event
            }
            break
        }
```

شکل 12- تابع addAccessCountAndUpdateCache

```
            higherAccess, _ := cache.accessCount.Load(cache.events[i-1].ID)
            lowerAccess, _ := cache.accessCount.Load(event.ID)
            if lowerAccess.(int) > higherAccess.(int) { // swap
                cache.events[i] = cache.events[i-1]
                cache.events[i-1] = event
            }
            break
        }
        if i == len(cache.events)-1 { // last element
            higherAccess, _ := cache.accessCount.Load(ev.ID)
            lowerAccess, _ := cache.accessCount.Load(event.ID)

            if lowerAccess.(int) > higherAccess.(int) { // replace
                cache.events[i] = event
            }
        }
    }
}
```

شکل 13- ادامه تابع addAccessCountAndUpdateCache

در ادامه ما از یک تابع دیگر به نام findEvent استفاده کرده ایم که برای اینکه ببینیم یک event در cache وجود دارد یا خیراین را کال میکنیم و اگر نبود cache miss اتفاق میافتد.