Piscine Golang - Day05

Created @October 24, 2021 11:28 PM

요약 : 이 문서는 API 설계 및 HTTP 학습을 다룹니다.

서브젝트 오류 문의 : bigpel66@icloud.com

Contents

Chapter I - General Rules

- 모든 Exercise들은 Golang 1.17로 진행하며 테스트 코드와 함께 작성되어야 합니다. 작성된 테스트 코드들은 Exercise와 함께 제출되어야 합니다.
- 제출된 테스트 코드 내에 테스트 케이스는 반드시 존재해야 합니다.
- 적극적인 테스트 케이스는 권장되며, 특히 예외 케이스에 신경써야 합니다.
- 평가자는 자신이 생각했던 테스트 케이스가 존재하지 않는 경우, 피평가자에게 테스트 케이스를 요청할 수 있습니다. 이 때 피평가자의 코드가 동작하지 않으면 평가는 종료됩니다.
- 테스트 진행은 "testing" 패키지를 이용해야 하며, "testing"과 "github.com/stretchr/testify/assert" 둘 중 하나를 일관성 있게 사용해야 합니다.
- 테스트 진행 시 testing 객체를 이용한 벤치마킹 역시 적극 권장됩니다.
- Exercise의 제출은 동료 평가 및 Moulinette 채점으로 **PASS** 혹은 **FAIL**이 결정됩니다.
- Exercise들은 gofmt를 이용하여 포맷팅이 되어 있어야 합니다. 동료 평가 진행 시, 제출된 Exercise가 포맷팅이 되어 있지 않다면 **FAIL**입니다.
- Exercise들은 golangci-lint를 이용했을 때, 별다른 문제가 없다는 것이 검증되어야 합니다. golangci-lint에서 문제가 있는 부분이 발견되면 **FAIL**입니다.
- 동료 평가 진행을 위한 주석은 적극 허용됩니다. 각 함수 및 변수들에게 적용된 주석은 Golang에서 정식으로 지원하는 godoc을 이용하여 Document로 만들 수 있습니다.
- 컴파일은 **go build**를 이용하며, 이를 위해 **go mod**를 먼저 이해해야 합니다. **go mod**를 이용하지 않고 제출된 Exercise 역시 **FAIL**입니다.
- 제공된 서브젝트를 철저히 파악하여 Golang의 기초 지식을 모두 얻을 수 있도록 하십시오.
- 서브젝트에서 요구하는 내용들이 짧더라도, 이 내용들을 이해하고 예상한대로 작동되도록 시간을 쏟는 것은 굉장히 가치 있는 행동입니다. 다양한 시도를 권장합니다.
- *Go ? Ahead ! Write in Go.*

go mod를 이용하기 전에 GOPATH, GOROOT를 알아보십시오.

Chapter II - Preamble

많은 기업들이 Golang을 채택하여 현업에서 사용하고 있고, 이에 따라 Golang으로 제공하는 API들이 다수 있으며 회사에서 공식적으로 지원하지 않더라도 사용자들이 만들어서 이용하는 API도 다수 있다. 대표적으로 Docker, Slack, Discord, Notion, Trello 등 서비스를 Golang API로 이용할 수 있다. Golang에 익숙해졌다면, 이들을 이용한 개발도 가능할 것이니 적극 활용해보자. 특히 Golang의 Package 생태계는 다양하고 강력하므로 pkg.go.dev 웹 사이트와 친해지도록 하자.

client

Package client is a Go client for the Docker Engine API. For more information about the Engine API, see the documentation: https://docs.docker.com/engine/api/ You use the library by creating a client object and calling methods on it. The client can be created either from environment variables with

https://pkg.go.dev/github.com/docker/docker/client

slack

This is the original Slack library for Go created by Norberto Lopes, transferred to a Github organization. This library supports most if not all of the api.slack.com REST calls, as well as the

https://pkg.go.dev/github.com/slack-go/slack

discordgo

DiscordGo is a Go package that provides low level bindings to the Discord chat client API. DiscordGo has nearly complete support for all of the Discord API endpoints, websocket interface, and voice interface. If you would like to help the DiscordGo package please use this link to add the official DiscordGo test bot dgo

https://pkg.go.dev/github.com/bwmarrin/discordgo

notionapi

An API client for the Notion API implemented in Golang It supports all APIs for Notion API version 2021-08-16 $ go get github.com/jomei/notionapi Follow Notion's getting started guide to obtain an Integration Token.

https://pkg.go.dev/github.com/jomei/notionapi

trello

A #golang package to access the Trello API. Nearly 100% of the read-only surface area of the API is covered, as is creation and modification of Cards. Low-level infrastructure for features to

https://pkg.go.dev/github.com/adlio/trello

Chapter III

Exercise 00 : Dictionary!

Aa Name	Tags
Turn-in directory	ex00
Files to turn in	dict/dictionary.go , dict/dictionary_test.go , main.go , go.mod
Module name	ex00
Forbidden packages	None

어느 언어에서든 Map이라는 것을 익히고 활용하는데 가장 좋은 예시는 사전입니다.

사전의 기본적인 기능을 구현해보면서 Map을 학습하고 다양한 활용 방안에 대해 생각해볼 것입니다.

아래와 같은 조건들을 만족해야 합니다.

- 사전에 단어와 정의를 등록할 수 있어야 합니다.
- 사전에 단어와 정의를 삭제할 수 있어야 합니다.
- 사전에 단어의 정의를 수정할 수 있어야 합니다.
- 사전에 단어의 정의를 찾아낼 수 있어야 합니다.

위와 같은 기본적인 기능을 지원하는 사전을 아래와 같은 함수의 원형으로 구현하십시오.

```
func Create(dictionary map[string]string, word, defnition string)
```

```
func Read(dictionary map[string]string, word string) string
```

```
func Update(dictionary map[string]string, word, definition string)
```

```
func Delete(dictionary map[string]string, word string)
```

사전을 구성하기 위한 자료구조의 선언 변수는 전역으로 두어 사용합니다.

전역 변수의 초기화는 main 함수 내에서 이뤄져야 하며, 이외의 전역 변수는 불허합니다.

```
❯ ./ex00
Creating a Word in Dictionary...
The Definition of Word Golang is Hello World

Updating a Word in Dictionary...
The Definition of Word Golang is is easy to learn

Deleting a Word in Dictionary...
The Definition of Word Golang has been deleted
```

위의 결과가 나오도록 main 함수를 구성해도 되고, 다른 결과가 나오도록 구성해도 됩니다.
다만 여러 예시가 잘 동작하도록 구성하십시오.

main 함수를 작성했다고 해서, 테스트 파일로부터 자유로운 것은 아닙니다.

테스트를 작성해가며 요구 사항들을 만족하는 코드를 작성하십시오.

작성한 함수들 중 몇몇의 생김새가 서로 닮았다면, 지극히 정상적인 상황입니다.

Chapter IV

Exercise 01 : Don't Scam with Definition.

Aa Name	Tags
Turn-in directory	ex01
Files to turn in	dict/dictionary.go , dict/dictionary_test.go , dict/dictionary_error.go , main.go go.mod
Module name	ex01
Forbidden packages	None

직접 정의한 사전이 만들어졌습니다!

만들어진 사전은 제 기능을 할 수 있지만, 많이 빈약합니다.

빈약한 점들을 보완하기 전에 사전이라는 타입에 대해서 정확하게 짚고 넘어갈 것입니다.

- 기존에 사전으로 이용한 타입을 Dictionary로 정의하십시오.
- 사전으로 이용한 함수들을 Dictionary의 메서드가 되도록 작성하십시오. 단, 이 때 각 함수들의 리시버는 포인터가 되어야 합니다.

이제 우리는 문제를 직면할 때입니다. 우리의 사전은 다음과 같은 문제가 있습니다.

- 단어를 생성할 때, 이미 생성된 쌍은 새로운 엔트리로 덮어쓰입니다.
- 단어를 읽어올 때, 정의를 추가하지 않은 쌍이 문제 없이 읽힙니다.
- 단어를 수정할 때, 생성되어 있지 않은 쌍은 추가되어버립니다.
- 단어를 삭제할 때, 없는 단어를 삭제하려는 시도가 있을 수 있습니다.

우리는 위와 같은 문제를 적절히 검출하고, 대처할 수 있도록 직접 Error를 정의할 것입니다. 특히 Create, Update, Delete 의 상황을 검출할 때는 Read를 기반으로 문제 상황을 찾아낼 수 있으므로, 이를 적극 이용해야 합니다.

Read 함수에서 결과로 나올 수 있는 Error는 다음과 같으며, 구현 시에 주어진 조건들을 만족해야 합니다.

```
ErrorNotFound = ... // "Read Error: Word is not in dictionary"
```

- Error는 반드시 열거형 상수로 정의해야 합니다.
- Error를 errors.New 혹은 fmt.Errorf를 이용하여 error 타입으로 이용할 수 있지만 진부합니다. 우리는 인터페이스를 배웠습니다. Error의 타입은 DictError가 되어 별도의 타입으로써 error로 이용될 수 있어야 합니다. 인터페이스의 특성을 잘 이해했다면 쉽게 만들어낼 수 있습니다. error는 내장 인터페이스 임을 고려하여 작성하십시오.
- 정의된 Error가 갖고 있는 내용은 주석에 표시된 문구와 동일하게 작성되어야 합니다.
- Read가 사용자가 정의한 Error를 반환함에 따라 함수의 반환 타입은 기존의 string에서 string과 error가 되어야 합니다.

Read를 이용했을 때 Error를 만들었기 때문에 Create, Update, Delete 함수에서는 Read를 이용하도록 만들 수 있고, Read의 결과에 따라서 각 함수에 맞는 Error가 나오도록 만들 수 있습니다.

Create 함수의 결과로 나올 수 있는 Error는 다음과 같으며, ErrorNotFound와 동일한 조건으로 구현되어야 합니다. Read 함수와 마찬가지로 Create 함수의 원형도 변경되어야 합니다.

```
ErrorAlreadyExist = ... // "Create Error: Word already exists in dictionary"
```

Update 함수의 결과로 나올 수 있는 Error는 다음과 같으며, ErrorNotFound와 동일한 조건으로 구현되어야 합니다. Read 함수와 마찬가지로 Update 함수의 원형도 변경되어야 합니다.

```
ErrorDoesNotExist = ... // "Update Error: Word does not exist in dictionary"
```

Delete 함수의 결과로 나올 수 있는 Error는 다음과 같으며, ErrorNotFound와 동일한 조건으로 구현되어야 합니다. Read 함수와 마찬가지로 Delete 함수의 원형도 변경되어야 합니다.

```
ErrorNotRemovable = ... // "Delete Error: Word cannot be removed from dictionary"
```

```
❯ ./ex01
Creating a Word in Dictionary...
The Definition of Word Golang is Hello World

Updating a Word in Dictionary...
The Definition of Word Golang is is easy to learn

Deleting a Word in Dictionary...
The Definition of Word Golang has been deleted
```

위의 결과가 나오도록 main 함수를 구성해도 되고, 다른 결과가 나오도록 구성해도 됩니다. 다만 여러 예시가 잘 동작하도록 구성하십시오.

마지막으로, 테스트 코드를 작성할 때 각 함수의 기능을 검증하는 것뿐만 아니라 Error가 가진 내용이 적절하게 검출되는지도 확인되도록 만들어야 합니다. Error를 검증하는 것도 중요하고, 그 과정에서 열거형 상수의 필요성을 느낄 수 있어야 합니다. 또한 이전 챕터와 마찬가지로 상황에 대한 적절한 main 함수가 작성되어야 합니다.

Chapter V

Exercise 02 : Anyone can use ?!

Aa Name	Tags
Turn-in directory	ex02
Files to turn in	dict/dictionary.go , dict/dictionary_test.go , dict/dictionary_error.go , router/router.go , router/router_test.go , main.go , go.mod
Module name	ex02
Forbidden packages	None

우리는 정의된 사전을 적절하게 보완해냈습니다. 완벽하진 않지만, 충분히 다른 사람들도 이용할 법 합니다. 이에 따라 우리는 사전을 공개하여 다른 사람들도 사전을 이용하여, 정의했던 Create, Read, Update, Delete를 이용할 수 있도록 만들 것입니다.

맞습니다. 이전에 구현한 사전에 HTTP 기능을 추가할 것입니다.

HTTP는 RESTful API 형태로 제공되어야 합니다. 기본 "net/http"를 이용해도 되지만, 시간은 금이기 때문에 "github.com/gorilla/mux"를 이용하여 더욱 편리하게 RESTful API를 만들 것입니다. 자세한 설명은 아래 링크를 통해 파악할 수 있습니다.

> mux
>
> https://www.gorillatoolkit.org/pkg/mux Package gorilla/mux implements a request router and dispatcher for matching incoming requests to their respective handler. The name mux stands for "HTTP request multiplexer". Like the standard http.ServeMux, mux.Router matches incoming requests against a list of
>
> https://pkg.go.dev/github.com/gorilla/mux

"github.com/gorilla/mux"에서 제공되는 mux를 이용하여 http.Handler를 만드십시오. 함수의 원형은 다음과 같으며, Handler에 등록되어야 하는 Router는 아래와 같습니다.

```
func Handler() http.Handler
```

- "/dictionary" → Method: "GET" → Handler 이름: GetDict
- "/dictionary" → Method: "POST" → Handler 이름: PostDict
- "/dictionary" → Method: "PUT" → Handler 이름: PutDict
- "/dictionary" → Method: "DELETE" → Handler 이름: DeleteDict

위 Router를 구성하는데 필요한 조건들은 아래와 같습니다.

- 주고 받는 Request와 Response는 데이터이므로 Content-Type을 application/json으로 설정하고 통신할 수 있어야 합니다.
- GET은 정의를 찾으려는 단어를 WORD라는 QueryString으로 넘기도록 로직을 구성해야 합니다.
- POST와 PUT은 단어에 대한 정보를 넘길 때 Request의 Body를 이용해야 하며, 단어와 정의를 넘깁니다.
- DELETE 역시 Request의 Body로 단어를 넘깁니다.
- 단어와 정의의 한 쌍의 묶음을 나타내는 구조체를 추가로 선언할 수 있으며, 해당 구조체를 선언할 경우에는 소문자로 만들어야 합니다.

Request의 Body를 이용할 때, "encoding/json"을 이용합니다. json의 Encoder와 Decoder를 이용하면 편하게 객체를 운용할 수 있습니다.

Router의 로직 구성을 마무리 지을 때, StatusCode가 전달되도록 만들어야 합니다. 각 Router에 해당되는 StatusCode는 아래와 같습니다.

- "GET"의 성공 → StatusOK / "GET"의 실패 → StatusNotFound
- "POST"의 성공 → StatusCreated / "POST"의 실패 → StatusBadRequest
- "PUT"의 성공 → StatusOK / "PUT"의 실패 → StautsBadRequest
- "DELETE"의 성공 → StatusOK / "DELETE"의 실패 → StatusBadRequest

각 HTTP의 Method들은 내부적으로 사전의 기능으로 구현한 API를 호출하도록 만들어야 합니다. 기존에 구현해둔 API들은 Error를 반환하도록 만들어져 있으므로, HTTP의 Method 진행에서 이를 검출하여 제시된 상태 코드로 보내줄 수 있도록 구현해야 합니다.

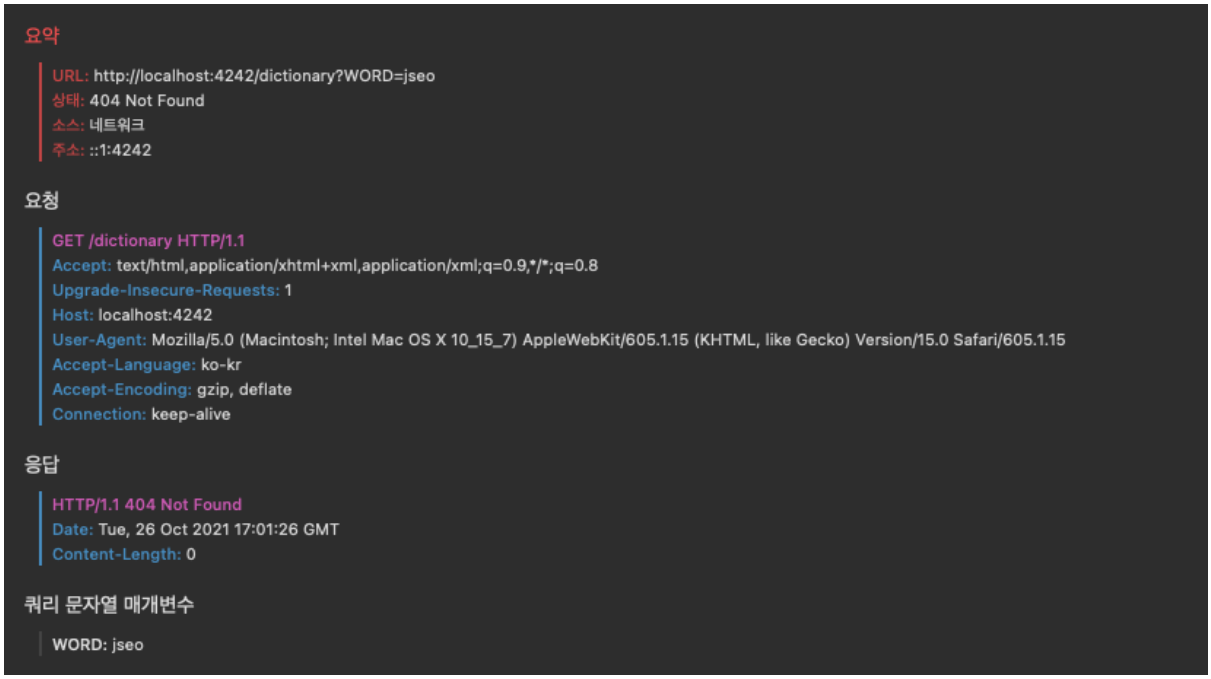
```
type Success struct {
  Success bool
}
```

각 HTTP의 Method들이 Response로 보내줘야 하는 사항들은 다음과 같습니다. 이 때 Success라는 구조체를 위와 같은 형태로 선언하여 이용하십시오.

- GET → 단어와 정의를 Response에 보내줍니다.
- 그 외 → 각 요청에 해당하는 작업이 성공했는지 여부를 Success 구조체로 보내줍니다.

구성된 Router를 갖고 있는 http.Handler가 준비되었다면, "net/http"의 ListenAndServe를 이용하여 서버를 구동하도록 작성하십시오. 이 때 포트는 반드시 4242로 이용해야 합니다. 또한 ListenAndServe에서 검출되는 error를 확인하여 panic을 발생하도록 만들고, main 함수에서는 이를 적절하게 복구할 수 있는 구문이 작성되어 우리가 제공하는 서비스가 종료되지 않도록 만들어야 합니다. 이 때 사전으로 이용하는 전역 변수의 위치는 기존과 다른 위치에 존재할 수도 있습니다. 사전의 초기화 위치를 잘 정하십시오.

이전 챕터들과 마찬가지로 각 함수의 기능을 검증하는 테스트 코드가 필요합니다. 이번 챕터의 테스트 코드에는 HTTP의 StatusCode를 테스트 하는 내용이 추가되어야 합니다. 특히 이번 챕터에서는 API 작성 시 만들어지는 테스트 코드가 가장 중요합니다.



main 함수에는 위에서 언급한 것처럼 반드시 panic과 recover 요소가 들어있어야 하며, 출력 문구는 자유롭게 구성해도 됩니다. 위와 같이 웹 사이트에 접속하여 결과를 확인해볼 수 있어야 합니다.

Chapter VI

Exercise 03 : I want to know what you're doing.

Aa Name	Tags
Turn-in directory	ex03
Files to turn in	dict/dictionary.go , dict/dictionary_test.go , dict/dictionary_error.go , router/router.go , router/router_test.go , main.go , go.mod
Module name	ex03
Forbidden packages	None

비록 로컬 상에서 작동하는 사전이 되었지만, 언제든 Public IP 주소를 얻는다면 제 기능을 할 수 있는 사전이 되었습니다. 하지만 메타 위키 사전처럼 되어 있는 우리의 사전은 누가 언제 무슨 행동을 했는지 알 수 없습니다. 기여해준 모든 사람들의 로그를 남겨 우리의 사전이 어떻게 업데이트 되어 왔는지, 사전 서비스를 이용하러 온 사용자들은 언제 요청을 해왔는지 알기 위해 Logger를 둘 생각입니다.

사전에 사용할 Logger는 "github.com/sirupsen/logrus"입니다. logrus에서는 5가지 레벨에 맞춰 기록을 하는 것이 가능합니다. 우리의 사전 서비스가 적절하게 기록을 할 수 있도록 Logger를 활용하십시오.

Console에 출력하도록 두는 것도 가능하고, 파일로 저장해두는 것도 가능합니다. 레벨 별 출력을 두었을 때, 일정 레벨 이상의 로그만 출력하도록 두는 것도 가능합니다. 또한 우리가 기록하는 내용들의 형식을 JSON으로 할 수도 있고, TEXT로 할 수도 있습니다.

```
var logger *logrus.Entry
```

위와 같은 전역 변수를 하나 더 허용합니다. 위 변수를 이용하여 적절하게 Logger를 활용하는 코드를 작성하십시오. logrus에서 지원하는 기능들이 방대하고 많기 때문에, 강력한 기능들을 꼼꼼하게 공부하는 것을 권장합니다.

Logger를 원하는 옵션으로 구성하여 초기화하고, 각 Router 별로 동작들이 적절하게 이행되는지 확인하는 코드를 구성해야 합니다. Request를 받았을 때 1회, Request를 처리했을 때 1회로 적어도 2회 이상의 기록이 보장되어야 합니다. 작성된 로그 출력 구문들은 동료 평가로 꼼꼼히 검증될 것입니다.