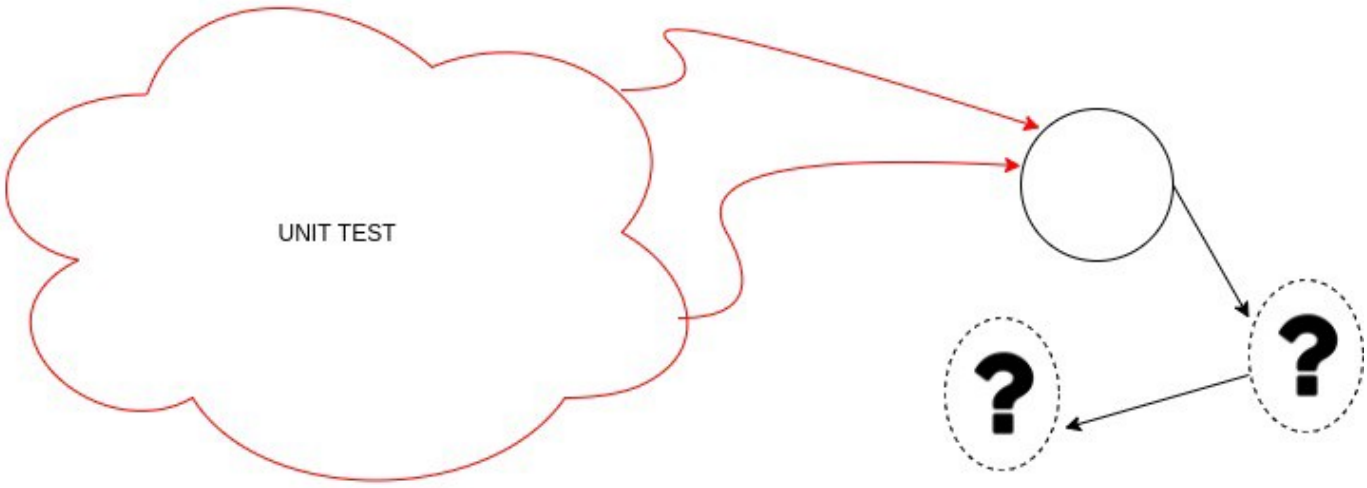
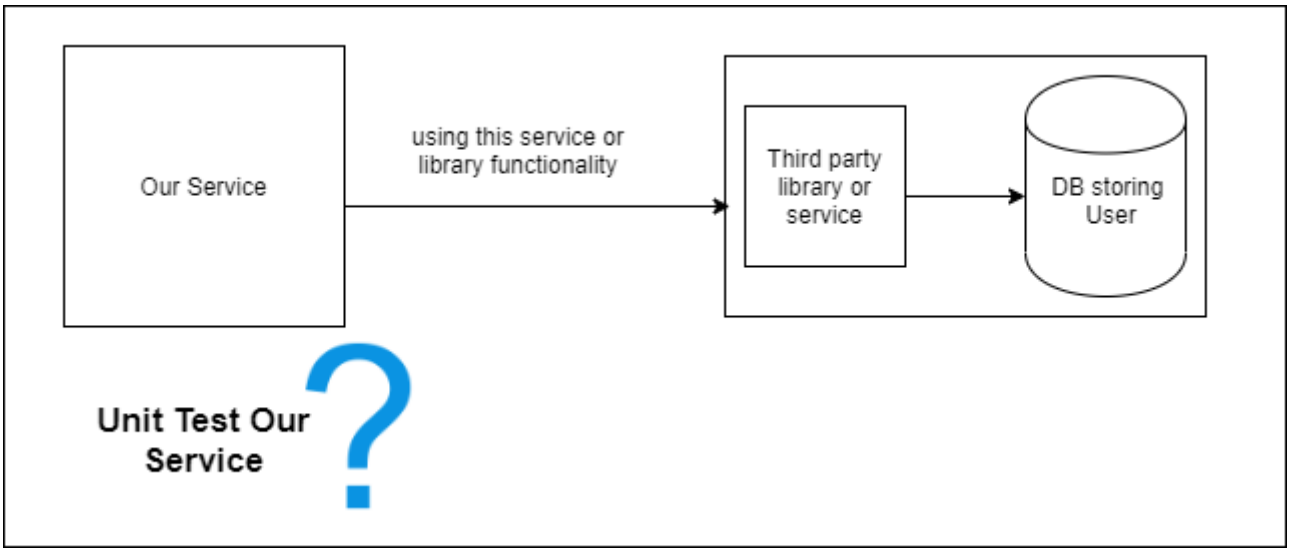
Если вы видите что-то необычное, просто сообщите мне.

Как мокать? Go способ.



У Go есть встроенный фреймворк тестирования предоставленный `testing` пакетом, это позволяет писать тесты проще, но тогда как мы пишем более сложные тесты которые требуют моков? В этой статье, мы изучим как взять преимущества структур и интерфейсов в Go, чтобы смокать любой сервис или библиотеку которую используем, без использования сторонних инструментов и библиотек. Начнем с определения нашей системы для понимания того. что мы будем делать с тестом и моком.

Система.



Наша система имеет 2 компонента:

- "Наш сервис" который у нас есть и мы создаем
- "Сторонние сервисы и библиотеки" которые взаимодействуют с базой данных и мы используем для работоспособности нашего сервиса. Теперь так как мы строим "Наш сервис", мы хотим написать независимую единицу для "Нашего сервиса" но так как мы используем функционал стороннего сервиса или библиотеки в нашем сервисе, если мы тестируем без моков, мы будет производить интеграционное тестирование, что иногда сложно и более временно затратно. Для демонстрации, мы напишем простую библиотеку которая проверяет существует ли пользователь внутри map. Наш сервис будет нужен для хранения бизнес логики, и это станет сторонней библиотекой внутри нашей стистемы.

Код сторонней библиотеки

```
package userdb

// db act as a dummy package level database.
var db map[string]bool

// init initialize a dummy db with some data
func init() {
	db = make(map[string]bool)
	db["ankuranand@dummy.org"] = true
```

```
    db["anand@example.com"] = true
}

// UserExists check if the User is registered with the provided email.
func UserExists(email string) bool {
    if _, ok := db[email]; !ok {
        return false
    }
    return true
}
```

Сервис

Регистрация пользователя использует сторонний код, для проверки наличия пользователя. Если пользователь не существует, то сервис просто возвращает ошибку,и в обратном случае выполняет обычную логику.

```
package simpleservice

import (
    "fmt"
    "log"

    "github.com/ankur-anand/mocking-demo/userdb"
)

// User encapsulate a user in the system.
type User struct {
    Name     string `json:"name"`
    Email    string `json:"email"`
    UserName string `json:"user_name"`
}

// RegisterUser will register a User if only User has not been previously
// registered.
func RegisterUser(user User) error {
    // check if user is already registered
```

```
	found := userdb.UserExists(user.Email)
	if found {
		return fmt.Errorf("email '%s' already registered", user.Email)
	}
	// carry business logic and Register the user in the system
	log.Println(user)
	return nil
}
```

```
package simpleservice

import "testing"

func TestCheckUserExist(t *testing.T) {
	user := User{
		Name:     "Ankur Anand",
		Email:    "anand@example.com",
		UserName: "anand",
	}

	err := RegisterUser(user)
	if err == nil {
		t.Error("Expected Register User to throw and error got nil")
	}
}
```

Если мы посмотрим на функцию `theRegisterUser`. Она вызывает Функцию `userdb.UserExist` которая является для нас стороннеё библиотекой и мы не можем протестировать нашу `RegisterUser` функцию без её вызова.

Моки

Попробуем это исправить моками.

> Мок-объекты соответствуют требованиям к интерфейсу и заменяют более сложные настоящие объекты.

Мок-объекты соответствуют требованиям к интерфейсу.

Для этого нам нужно переделать код нашего сервиса. Первое, мы должны определить требования интерфейса, для того чтобы реализовать наш мок. В нашем случае, нам нужен интерфейс который внутренний для пакета который предоставляет тестирование существует пользователь или нет.

```
// registrationPreChecker validates if user is allowed to register.
type registrationPreChecker interface {
    userExists(string) bool
}
```

Реализация интерфейса.

`userExists` функция нашего нового интерфейса просто оборачивает вызова настоящего вызова к стороннему сервису.

```
type regPreCheck struct {}
func (r regPreCheck) userExists(email string) bool {
    return userdb.UserExist(email)
}
```

Теперь создадим, переменную на уровне пакета типа `registrationPreChecker` и присвоим экземпляр `regPreCheck` внутри `init` функции.

```
var regPreCond registrationPreChecker

func init() {
  regPreCond = regPreCheck{}
}
```

Так как `regPreCond` является типом `registrationPreChecker` который проверяет существования пользователя, мы можем использовать `RegisterUser` фнукцию. Поэтому вместо прямого

вызова функции `userdb.UserExist` внутри функции `RegisterUser` мы вызовем через реализацию интерфейса.

```
// check if user is already registered
found := regPreCond.userExist(user.Email)
```

Изменный код:

```
package unitservice

import (
	"fmt"
	"log"

	"github.com/ankur-anand/mocking-demo/userdb"
)

// User encapsulate a user in the system.
type User struct {
	Name     string `json:"name"`
	Email    string `json:"email"`
	UserName string `json:"user_name"`
}

// Mock objects meet the interface requirements of,
// and stand in for, more complex real ones
type registrationPreChecker interface {
	userExists(string) bool
}

type regPreCheck struct{}

func (r regPreCheck) userExists(email string) bool {
	return userdb.UserExist(email)
}

var regPreCond registrationPreChecker

func init() {
```

```
	regPreCond = regPreCheck{}
}

// RegisterUser will register a User if only User has not been previously
// registered.
func RegisterUser(user User) error {
	// check if user is already registered
	found := regPreCond.userExist(user.Email)
	if found {
		return fmt.Errorf("email '%s' already registered", user.Email)
	}
	// carry business logic and Register the user in the system
	log.Println(user)
	return nil
}
```

Если мы запустим test опять он пройет так как мы не трогали поведение нашей функции.

Теперь давайте посмотрим как заставить наше тестирование проходить с помощью моков.

Написание моков

Для начала посмотрим на полный код:

```
package unitservice

import "testing"

// This helps in assigning mock at the runtime instead of compile time
var userExistsMock func(email string) bool

type preCheckMock struct{}

func (u preCheckMock) userExists(email string) bool {
	return userExistsMock(email)
}

func TestRegisterUser(t *testing.T) {
```

```
	user := User{
		Name:     "Ankur Anand",
		Email:    "anand@example.com",
		UserName: "anand",
	}

	regPreCond = preCheckMock{}
	userExistsMock = func(email string) bool {
		return false
	}

	err := RegisterUser(user)
	if err != nil {
		t.Fatal(err)
	}

	userExistsMock = func(email string) bool {
		return true
	}
	err = RegisterUser(user)
	if err == nil {
		t.Error("Expected Register User to throw and error got nil")
	}
}
```

Мок объекты отвечают требованиям интерфейса. Тут наш мок объект реализует `registrationPreChecker` интерфейс.

```
type preCheckMock struct{}

func (u preCheckMock) userExists(email string) bool {
   return userExistsMock(email)
}
```

Реализация мока возвращает `userExistMock` тип функции вместо прямого возвращения `true` или `false`. Это помогает в назначении мока во время работы вместо во время компиляции. Вы можете увидеть это в `TestRegisterUser` фукнции.

Их заменяют более сложные настоящие объекты

```
regPreCond = preCheckMock{}
```

Мы просто назначили нашему `regPreCond` тип `registrationPreChecker`, который проверяет есть ли пользователь или нет в нашей реализации мока во времы выполнения теста. Это можно увидеть в `TestRegisterUser` функции.

```
func TestRegisterUser(t *testing.T) {
  user := User{
    Name:     "Ankur Anand",
    Email:    "anand@example.com",
    UserName: "anand",
  }

  regPreCond = preCheckMock{}
  userExistsMock = func(email string) bool {
    return false
  }

  err := RegisterUser(user)
  if err != nil {
    t.Fatal(err)
  }

  userExistsMock = func(email string) bool {
    return true
  }
  err = RegisterUser(user)
  if err == nil {
    t.Error("Expected Register User to throw and error got nil")
  }
}
```

Но мы еще не закончиил!

Мы сказали что будет делать через "Способ GO", да, но пока есть проблема, для этого нужно провести рефакторинг.

Мы использовали глобальную переменную и события, которые мы не обновляем переменную во время реального выполнения, это должно сломать паралельный тест. Есть несколько способов, чтобы это исправить. В нашем случае, мы собираемся передать `registrationPreCheker` как зависимость нашей функции и представим новую функцию конструктора которая будет создавать по умолчанию `regitrationPreCheck` тип, который может быть использован во время настоящей работы, и так как мы передаем его как зависимость, мы можем передать нашу реализацию мок как параметр, в этом случае.

```
func NewRegistrationPreChecker() RegistrationPreChecker {
  return regPreCheck{}
}

// RegisterUser will register a User if only User has not been previously
// registered.
func RegisterUser(user User, regPreCond RegistrationPreChecker) error {
  // check if user is already registered
  found := regPreCond.userExists(user.Email)
  if found {
    return fmt.Errorf("email '%s' already registered", user.Email)
  }
  // carry business logic and Register the user in the system
  log.Println(user)
  return nil
}
```

Let’s modify our test code too. So instead of modifying the package level variable, we are now explicitly passing it as a dependency inside our `RegisterUser` function.

```
func TestRegisterUser(t *testing.T) {
  user := User{
    Name:     "Ankur Anand",
```

```
    Email:    "anand@example.com",
    UserName: "anand",
  }

  regPreCond := preCheckMock{}
  userExistsMock = func(email string) bool {
    return false
  }

  err := RegisterUser(user, regPreCond)
  if err != nil {
    t.Fatal(err)
  }

  userExistsMock = func(email string) bool {
    return true
  }
  err = RegisterUser(user, regPreCond)
  if err == nil {
    t.Error("Expected Register User to throw and error got nil")
  }
}
```

Выводы

Чтож. так как этоо не единственный способ написания моков в GO, но с этим я надуюсь у вас появиться общее понимание моков и как использовать структуры и интерфейсы в GO чтобы мокать любые артефакты которые вам понадобятся, без внешних блиблиотек.

Revision #6
Created 17 January 2022 07:53:29 by gasick
Updated 16 April 2023 19:30:03 by gasick