GO-06 02: Моки, стабы и генерация через GoMock

Описание:

Давайте поговорим о таком понятии, как моки (Mocks) и стабы (Stubs). Все эти понятия можно для начала объединить одним термином Fakes или подделка реализации.
Наши функции редко бывают чистыми и редко работают без зависимостей. Зачастую они либо лезут в базу, либо отправляют request на сторонний сервис, либо вызывают другие наши функции, которые, в свою очередь, работают с разными зависимостями.
В написании тестов для таких функций перед нами сразу же встает вопрос:

- А что будет, если тестовая база ляжет?
- А что будет, если сторонний сервис будет отвечать дольше обычного?

И так далее. Если мы задаемся такими вопросами, это означает, что наши тесты перестали быть Unit, они вышли на уровень выше. Но наша задача в Unit-тестах - тестировать наш код, а не то, как он работает с другими частями системы.
Для тестирования такого кода мы можем воспользоваться таким инструментом, как моки.
Моки
Давайте взглянем на следующий пример.

```
type User struct {
    SessionID string
    Name      string
    Pass      string
}

type Session struct {
    ID     string
    UserID int
}

type Storage interface {
    RecoverSession(id string) (*Session, error)
    CreateSession(name string, pass string) (*Session, error)
}

func InitSession(user User, storage Storage) (*Session, error) {
    session, err := storage.RecoverSession(user.SessionID)
    if err != nil {
```

```
        return nil, err
    }

    if session == nil {
        return storage.CreateSession(user.Name, user.Pass)
    }

    return session, nil
}
```

Как можно заметить, в функции InitSession есть одна зависимость, которая будет мешать нам ее тестировать, - это storage. Ведь он вполне может быть, например, базой данных или клиентом другого сервиса, все зависит от реализаций интерфейса Storage. Чтобы тестировать функцию InitSessions, не задумываясь о ее зависимостях, хорошо бы подменить Storage на что-то такое, что мы сможем контролировать во время теста. Например, создать свое хранилище, реализующее интерфейс Storage, и подкладывать его в тесты.
В этом и заключается смысл моков. Мы подменяем функционал, который не можем контролировать (БД, клиенты, брокеры сообщений, и т.д.), на абсолютно такой же с точки зрения их интерфейса, но с другой реализацией, которую мы можем контролировать (управлять ее поведением).

```
type fakeStorage struct {
    createReturns  fakeStorageReturns
    recoverReturns fakeStorageReturns
}

type fakeStorageReturns struct {
    Session *Session
    Err     error
}

func (fs *fakeStorage) RecoverSession(id string) (*Session, error) {
    return fs.recoverReturns.Session, fs.recoverReturns.Err
}

func (fs *fakeStorage) CreateSession(name string, pass string)
(*Session, error) {
    return fs.createReturns.Session, fs.createReturns.Err
}

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

```
    t.Run("create session success", func(t *testing.T) {
        req := require.New(t)
        fs := fakeStorage{
            createReturns:  fakeStorageReturns{Session: Session{ID
"session_id", UserID: 1}, Err: nil},
            recoverReturns: fakeStorageReturns{Session: nil, Err:
nil},
        }

        session, err := InitSession(User{Name: "ivan", Pass:
"secret"}, fs)
        req.NoError(err)
        req.Equal("session_id", session.ID)
    })
}
```

Таким образом, передавая нужные нам для конкретного теста параметры в fakeStorage, мы можем добиваться различных результатов теста без взаимодействия с настоящим хранилищем.

Стабы

Стаб - это имитация заданного состояния. Главное отличие стаба от мока в том, что стаб ничего не проверяет (не то, сколько раз была вызвана функция, не то, с какими аргументами она была вызвана, и т.д), он лишь имитирует статическое состояние какого-либо объекта. Мы на протяжении всего теста знаем, что ожидать от стаба, тогда как поведение мока мы настраиваем от кейса к кейсу.

Отредактируем пример так, чтобы он был больше похож на стаб.

```
type fakeStorage struct {
}

func (fs *fakeStorage) RecoverSession(id string) (*Session, error) {
    return nil, nil
}

func (fs *fakeStorage) CreateSession(name string, pass string)
(*Session, error) {
    return Session{ID "session_id", UserID: 1}, nil
}

func TestInitSession(t *testing.T) {
    req := require.New(t)
    fs := fakeStorage{}
```

```
    cases := map[string]struct{
        user User
    }{
        "user exists":    {user: User{Name: "ivan", Pass: "secret"}},
        "without secret": {user: User{Name: "ivan"}},
        "without name":   {user: User{Pass: "secret"}},
        "empty":          {user: User{}},
    }

    for name, cs := range cases {
        t.Run(name, func(t *testing.T) {
            session, err := InitSession(cs.user, fs)
            req.NoError(err)
            req.Equal("session_id", session.ID)
        })
    }
}
```

Теперь в любом тесте наш стаб стораджа будет выдавать одно и то же состояние (одни и те же значения сессии и ошибки).
Чаще всего стабы подходят в тех тестах, где от поведения функции, которую мы хотим подменить, ничего не зависит.

- Например, функция просто должна у себя что-нибудь сохранить (инсерт в БД, например, и нам все равно, что будет туда сохраняться, ведь с самой БД мы не работаем, главное, чтобы функция insert не выдавала ошибку на протяжении теста).
- Обратным примером может быть функция, которая возвращает что-то статическое, например, конфиг приложения.

GoMock
Создавать функции моков/стабов так, как мы делали выше, не совсем удобно, потому что:

- Все моки/стабы надо поддерживать и дописывать самостоятельно.
- Нужно учитывать много важного функционала (например, запуск мока с одинаковыми return параметрами несколько раз подряд и многого другого), а это уже превращается в написание собственного фреймворка.

Есть и хорошие новости: за нас такой фреймворк уже написали! И называется он GoMock.
Работая через кодогенерацию, фреймворк генерирует заглушки из интерфейсов пакета, которые в дальнейшем мы можем использовать в качестве моков или стабов.
После установки GoMock мы можем генерировать моки для наших пакетов двумя различными путями:

1. Запуск через cli утилиту mockgen, где указываем, откуда брать интерфейсы для моков -source=[path/to/file.go], куда их складывать -destination=./path/to/mocks_dir и как будет называться данный пакет моков -package=mock_package_name.

mockgen -source=./app/app.go -destination=./app/generated/mocks/app_mocks.go -package=appMock

2. Если у нас много разных пакетов, для которых нужны моки, хорошим вариантом будет создать пакет gomock и в нем перечислить все запуски команды mockgen для всех пакетов.

Пример содержимого пакета gomock:

```
package gomock

//go:generate mockgen -source=./path/to/first/package.go
-destination=./mocks/first_package_mock.go -package=firstPackageMocks
//go:generate mockgen -source=./path/to/second/package.go
-destination=./mocks/second_package_mock.go
-package=secondPackageMocks
```

Затем просто запускаем генерацию командой go generate ./....
Теперь отредактируем тесты с использованием gomock.

```
func TestInitSession(t *testing.T) {
    req := require.New(t)
    any := gomock.Any()

    mockCtrl := gomock.NewController(t)
    defer mockCtrl.Finish()

    storageMock := mocks.NewStorageMock(mockCtrl)

    t.Run("create session success", func(t *testing.T) {
        storageMock.EXPECT().RecoverSession(any).Return(nil,
nil).Times(1)
        storageMock.EXPECT().CreateSession(any,
any).Return(&Session{ID "session_id", UserID: 1}, nil).Times(1)

        session, err := InitSession(User{Name: "ivan", Pass:
"secret"}, storageMock)
        req.NoError(err)
        req.Equal("session_id", session.ID)
    })
}
```

Теперь мы можем заранее сгенерировать моки в тесте, и они будут работать как стабы, или редактировать моки под каждый случай, указывая, с какими аргументами функции вызываются, сколько раз вызываются и что возвращают.

Полезные ссылки:

- gomock framework
- Тестирование для чайников

Задание:

В этом задании нам предстоит написать тесты к функции PostCount. Эта функция через http запрос достает посты пользователей какого-нибудь ресурса (представим, например, что этот ресурс - habr). API этого ресурса может работать, а может и не работать, поэтому тестировать эту функцию напрямую нельзя. Нам нужно замокать зависимости этой функции и протестировать только наш код.

1. В вашем проекте module06 сделайте новую ветку module06_02.
2. Проанализируйте функцию PostCount из пакета ./internal/app/processors/counter, какие зависимости есть и как они работают.
3. Сгенерируйте моки для клиента постов при помощи GoMock. Destination каталог должен быть ./test/gomock/mocks/postmock.
4. Создайте файл post_counter_test.go в каталоге ./internal/app/processors/counter.
5. Напишите тесты с использование сгенерированных моков.
6. В качестве ответа пришлите ссылку на merge request в ветку master вашего проекта ветки module06_06, в которой должны быть:
    - Моки сгенерированные в папку ./test/gomock/mocks/postmock.
    - Должен быть тест с использованием сгенерированных моков.