GeekBrains

Лучшие практики разработки Go-приложений

Принципы структурирования Go-приложений

На этом уроке

1. Рассмотрим популярные принципы структурирования Go-приложений, основные их особенности, достоинства и недостатки.
2. Научимся создавать чистую и понятную структуру приложения.
3. Научимся работать с абстракциями.

Оглавление

[На этом уроке](#)

[Теория урока](#)

[Общая структура приложения и принципы проектирования](#)

- [“Плоская” структура](#)
- [Многоуровневая структура](#)
- [Группировка по модулям](#)
- [Группировка по контексту(DDD)](#)
- [Гексагональная архитектура](#)
- [Монорепозитории](#)
- [Обзор project-layout](#)

[Абстракции в Go](#)

- [Паттерн “Accept interfaces return structs”](#)

[Структура типового веб-сервиса](#)

[Структура библиотеки](#)

[Практическая часть](#)

[Глоссарий](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Теория урока

В данном разделе будут рассмотрены основные подходы к структурированию Golang приложений, такие как:

- “плоская” структура
- многоуровневая структура(группировка по специфическим функциям)

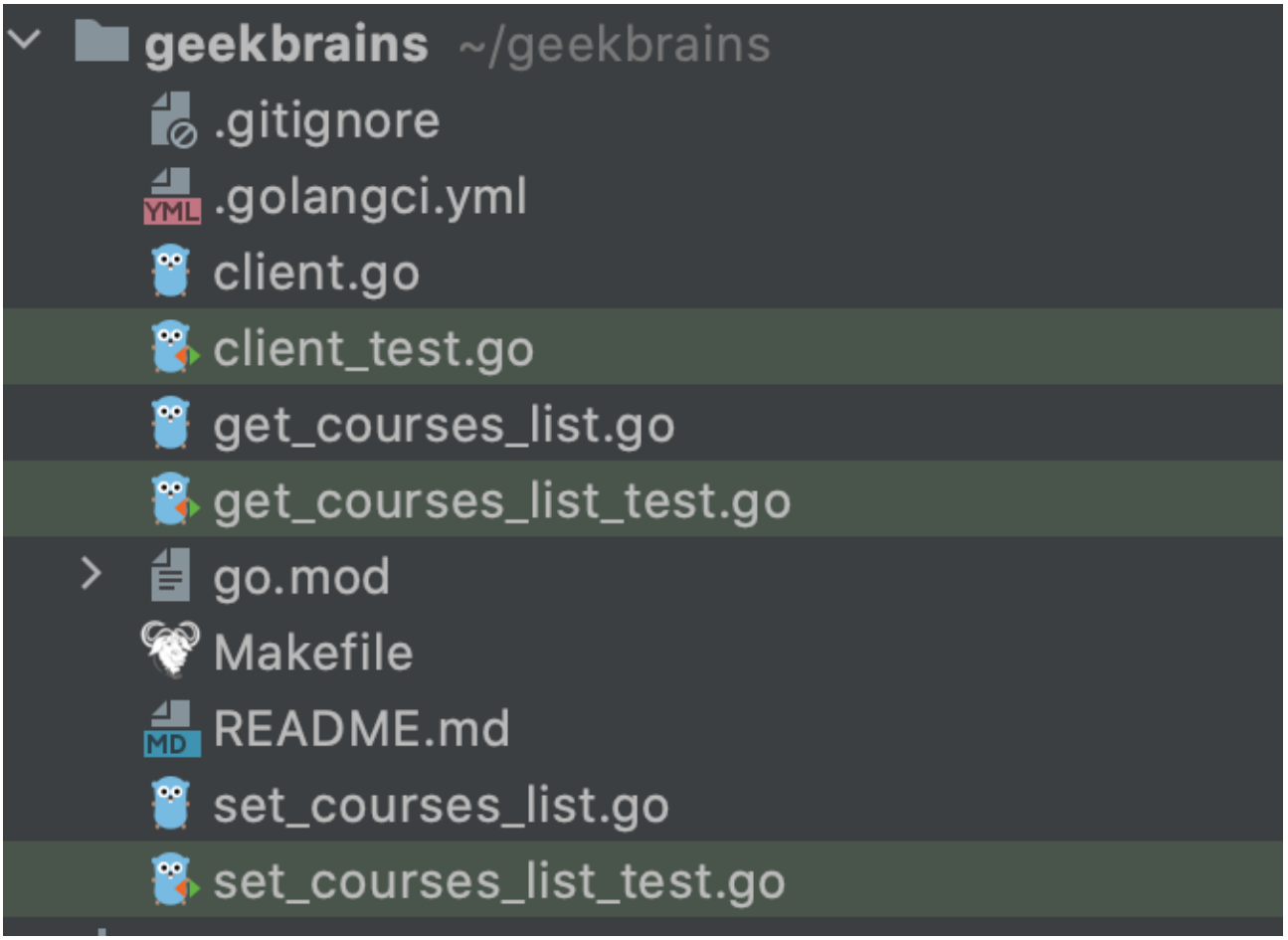
- группировка по модулям/контексту
- гексагональная архитектура

Помимо вышеперечисленного постараемся понять, в каких случаях будет предпочтительнее выбрать один подход, а в каких случаях другой. Рассмотрим вопрос построения читаемых, удобных в поддержании библиотек, а также пример структуры проекта на основе project-layout.

Общая структура приложения и принципы проектирования

“Плоская” структура

Под “плоской” мы будем понимать структуру, в которой файлы располагаются в каталоге проекта сплошным полотном. Рассмотрим, например, структуру вымышленного клиента к порталу geekbrains. Мы хотим, чтобы наш пакет умел получать список курсов, устанавливать список курсов. В случае плоской структуры пакет может выглядеть следующим образом:



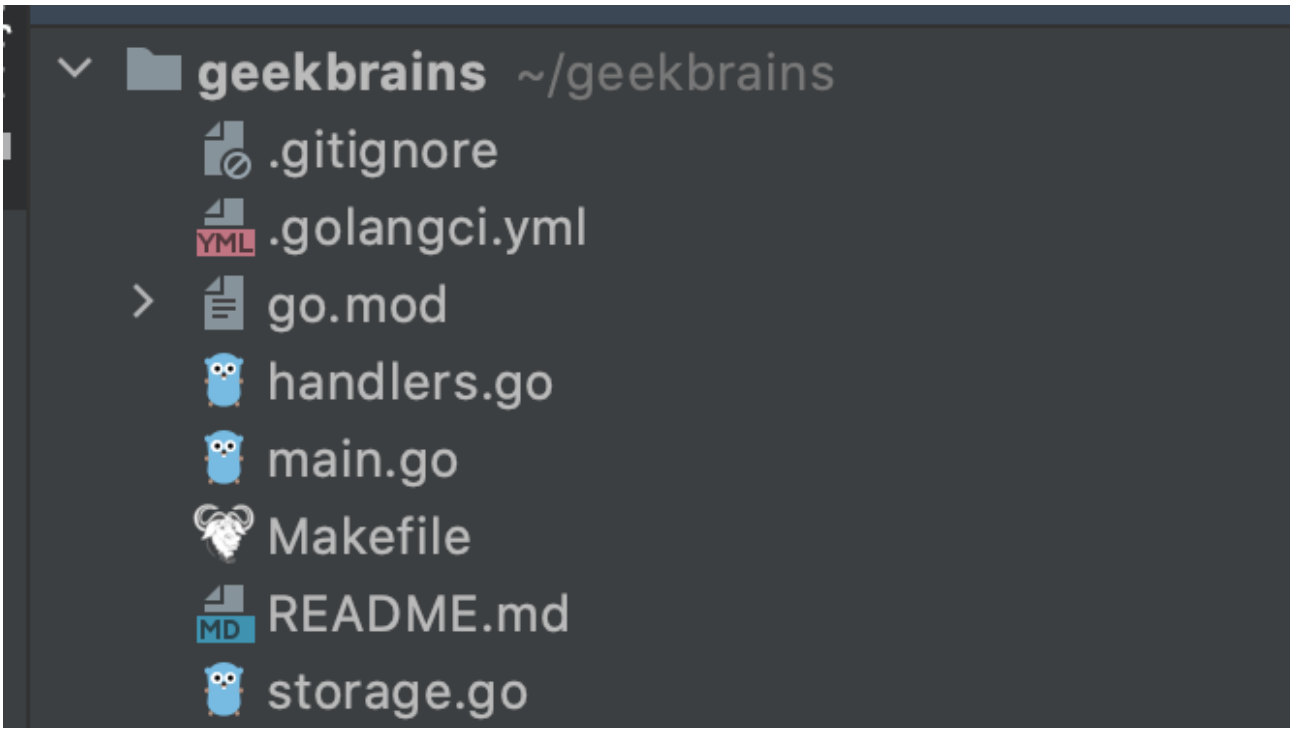
В этом случае можно не усложнять структуру, так как у нашего клиента пара функций, ничего не осложняет чтение, понимание и навигацию по пакету.

Также данный подход может быть уместен при написании простого микросервиса с несколькими ручками, если у нас есть четкое понимание, что дальнейшего развития у проекта нет(может быть применимо к пет проектам) или же мы отдаем себе отчет, что нас и нашу команду такой подход к структурированию приложения устроит, и в случае усложнения логики мы готовы перестроить пакет.

На скриншоте ниже представлена структура простого микросервиса с тремя гошными файлами:

- main.go -- точка входа, инициализация сервера и прочее
- handlers.go -- реализация обработчиков
- storage.go -- логика работы с хранилищем.

На самом деле, никто не запрещает писать весь код в main.go, даже не дробя логику на файлы.



В итоге можно сделать вывод, что такой подход к структурированию уместен, если мы реализуем простой микросервис или библиотеку. Также стоит отметить, что при таком структурировании исключена возможность циклических зависимостей, за счет того, что весь код находится в одном пакете. Если наше приложение будет развиваться и расширяться, то такого рода структура может быть неудобна в поддержке, может усложниться чтение и навигация по пакету. Опыт показывает, что в таком случае будет удобнее воспользоваться другой структурой приложения, поговорим об этих структурах далее.

Примеры успешных проектов с плоской структурой:

1) gjson

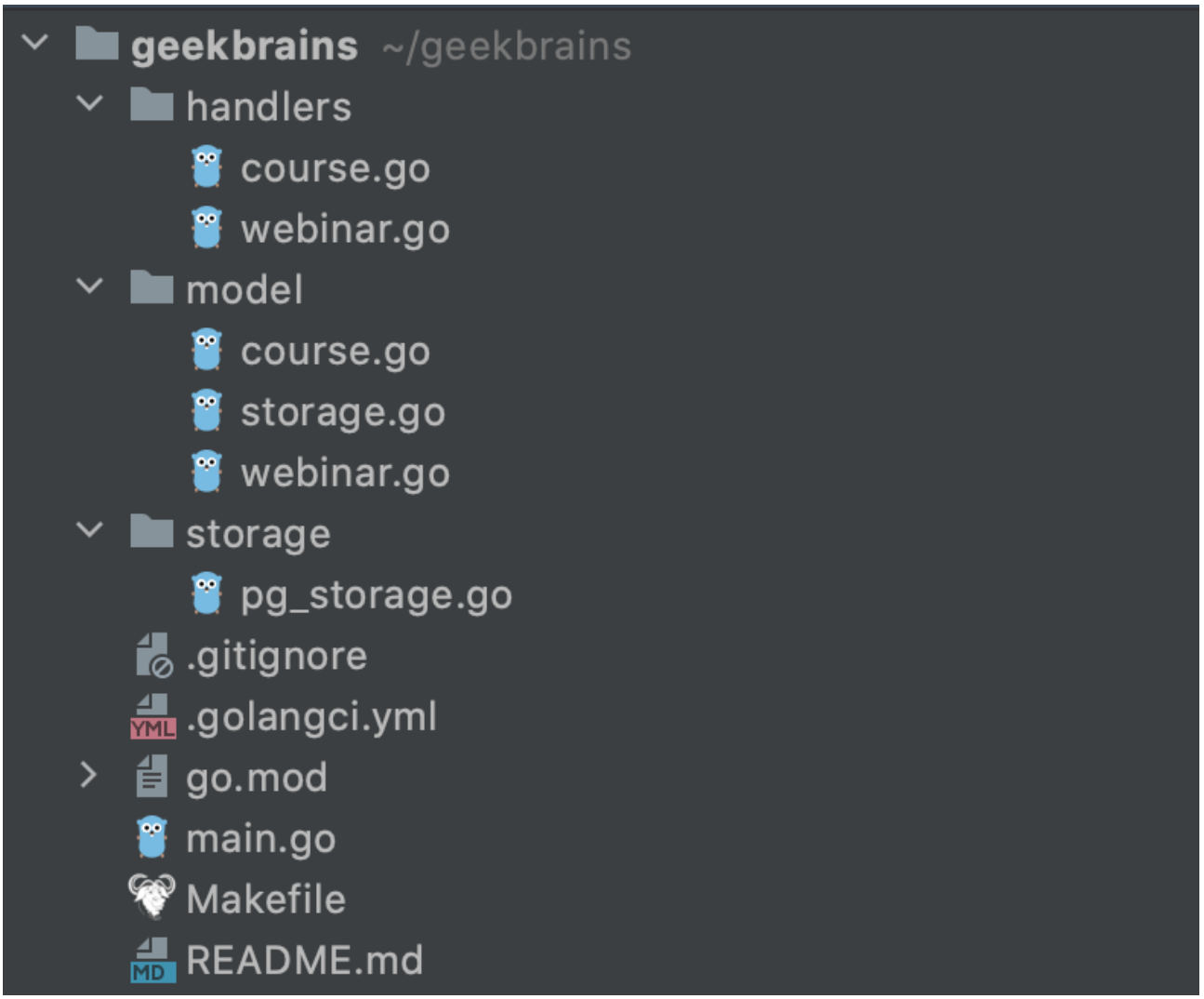
2) yaml

Многоуровневая структура

В многоуровневой структуре наше приложение разбивается на несколько уровней(слоев), например, знаменитый и широко используемый во фреймворках и иных языках программирования MVC паттерн, который включает в себя:

- уровень представления данных (View)
- уровень контроллера (Controller)
- уровень модели (Model)

На рисунке ниже изображен пример одного из подходов к многоуровневому структурированию проекта:



Каждый уровень отвечает за определенную функцию приложения, на первый взгляд все в этой структуре хорошо, однако здесь возможно возникновение циклических зависимостей, например, между пакетами storage и model.

Примеры проектов с MVC паттерном:

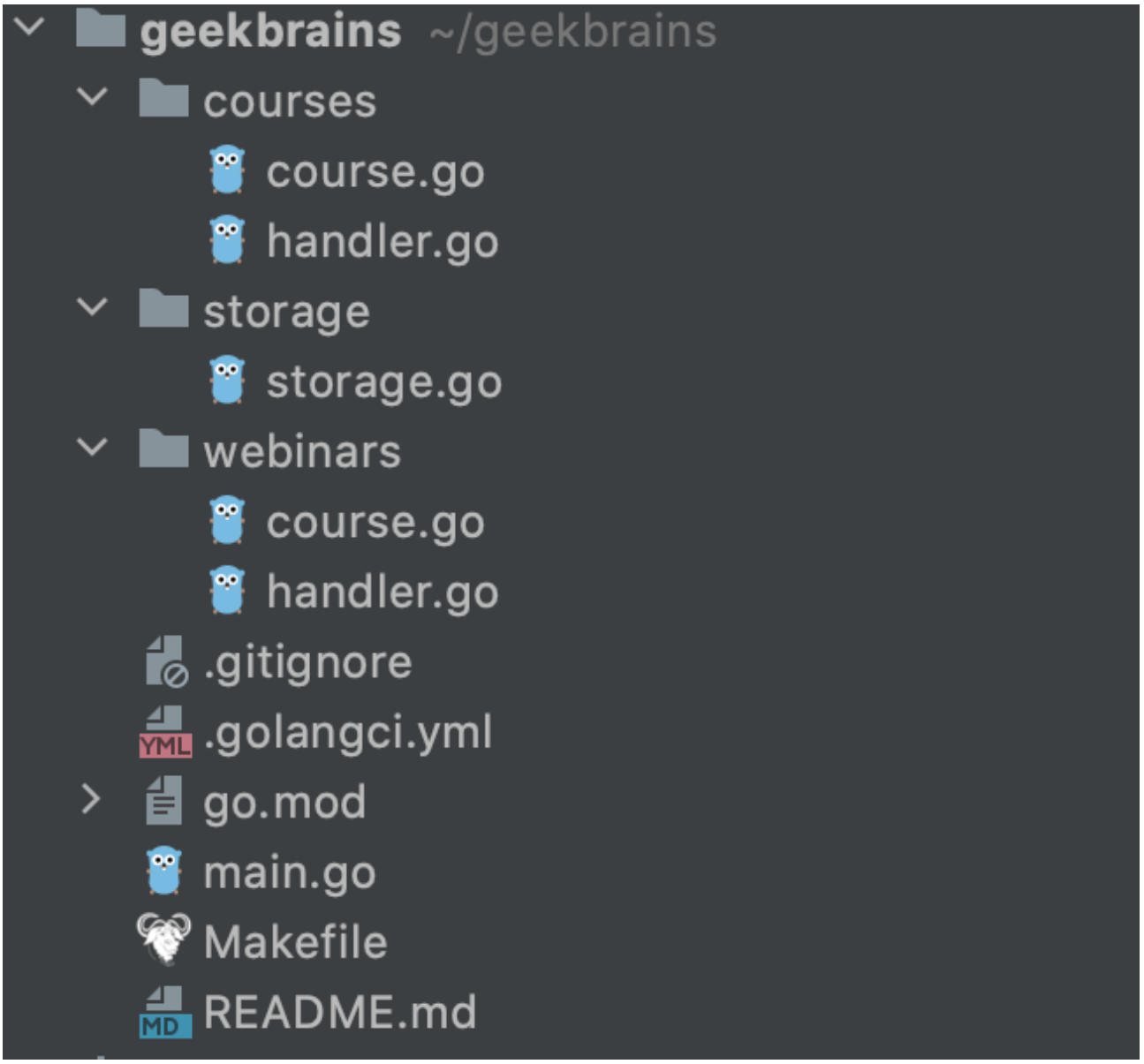
1) Go restful MVC

2) Пример MVC проекта на utron framework’е

Группировка по модулям

В группировке по модулям суть заключается в выделении пакетов, которые принадлежат, к примеру, одной сущности(ресурсу).

На скриншоте ниже представлен пример проекта, построенного благодаря группировке по модулям:



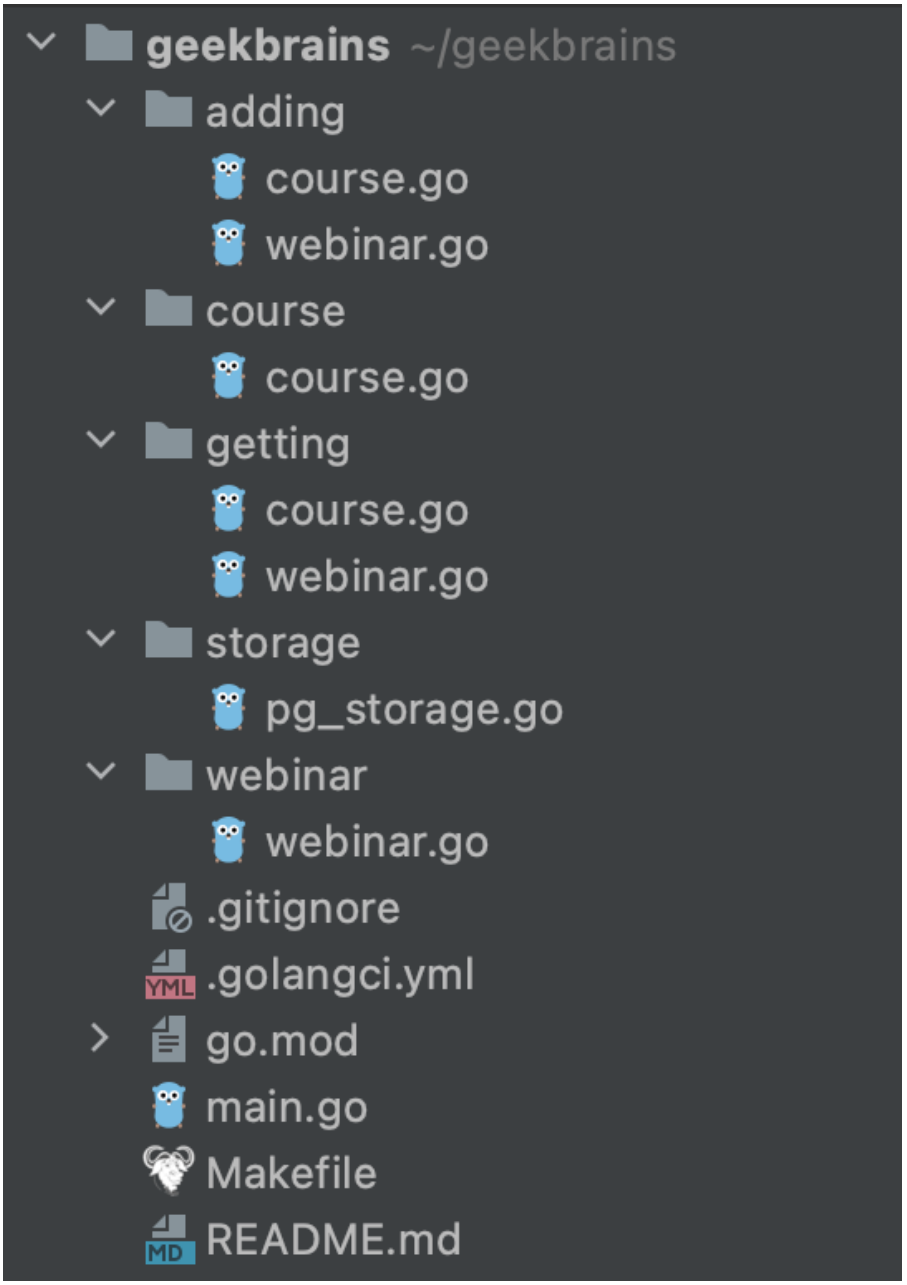
Здесь, как и в родительской многоуровневой структуре, возможны циклические зависимости.

Группировка по контексту(DDD)

Domain driven development(DDD или предметно ориентированное проектирование) — это набор принципов по проектированию вашего приложения. DDD не содержит каких-либо рекомендаций по расположению файлов или пакетов, подход предлагает задуматься о предметной области(домене) и бизнес логике вашего приложения еще до написания кода. В этом случае названия пакетов могут

называться не опираясь на то, что они содержат, а основываясь на действии с сущностью, которое совершается в данном пакете.

Например, следующим образом:



В DDD сущность в различных контекстах/сервисах может иметь разные характеристики. Например, сущность пользователя в контексте продаж важна с точки зрения стоимости его привлечения и подобных характеристик, если же рассматривать пользователя в контексте обращения в службу поддержки -- будут важны другие его характеристики. Данный подход позволяет гибко менять сущность для определенного контекста, не затрагивая при этом другой контекст.
Не лишним будет подчеркнуть, за счет того, что DDD не регламентирует расположение файлов или каталогов, вы можете применять данный подход и при плоской структуре приложения.

Примеры применения DDD в golang:

1) goddd

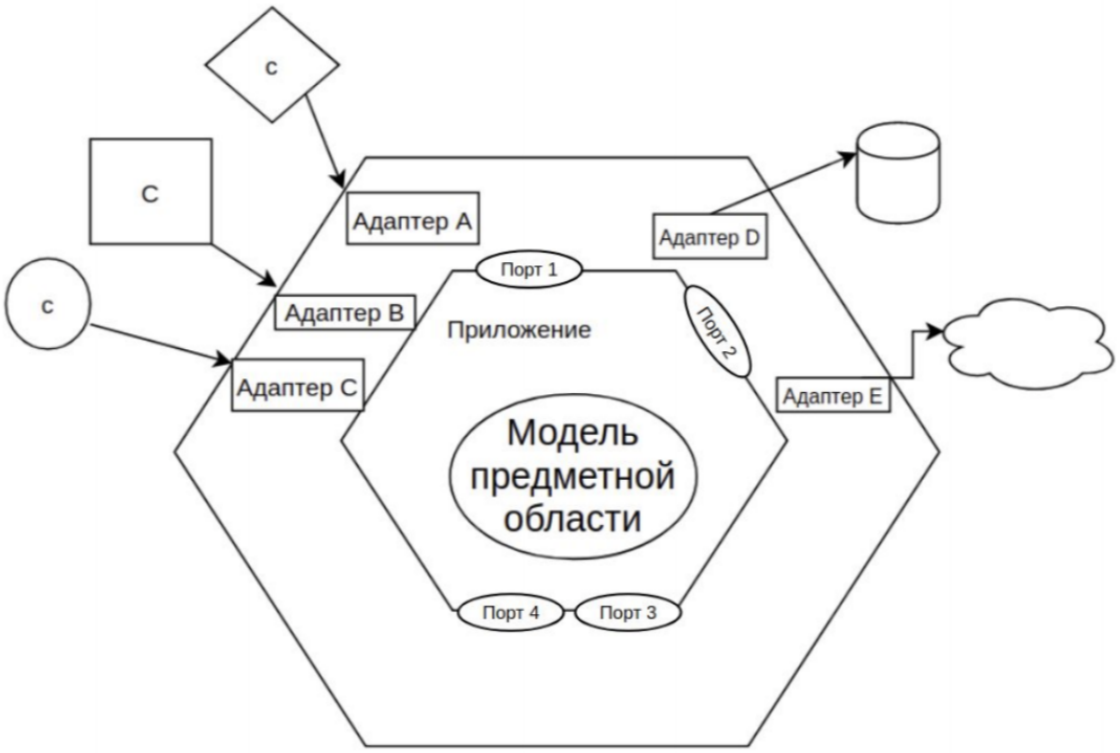
2) Kat Zien DDD example

3) Takashabe Go ddd sample

Гексагональная архитектура

Гексагональная архитектура позволяет изменить одну часть приложения, не затрагивая при этом другую часть. Например, вы можете использовать в своем приложении базу Postgres, и если через какой-то промежуток времени появится необходимость использовать нереляционную БД вместо Postgres -- это можно будет сделать не изменяя часть проекта, которая ответственна за бизнес-логику. И наоборот, если необходимо немного видоизменить бизнес-логику, изменения, вероятно, затронут лишь один пакет. Это достигается за счет того, что в гексагональной архитектуре в рамках нашего примера бизнес логика не зависит от реализации взаимодействия с базой, а зависит только лишь от абстракции(интерфейса), поэтому данный переход может быть реализован бесшовно, если пакет взаимодействия с нереляционной базой в нашем случае будет реализовывать тот же интерфейс. Аналогичная логика работает со сторонними сервисами и прочим. Примеры можно посмотреть в репозитории Kat Zien, докладе Kat Zien, а также в репозитории golang-api-showcase.

Ниже представлено схематическое изображение гексагональной архитектуры или, как ее еще называют, архитектура портов и адаптеров.



Под портами подразумеваются интерфейсы, а под адаптерами их реализация. Недостатком гексагональной архитектуры можно считать повторение интерфейсов в различных частях приложения. Например, интерфейс для работы с хранилищем может быть объявлен в пакете с бизнес-логикой приложения, а также в пакете с крон-джобой, которая взаимодействует с хранилищем.

Не лучшим решением будет использовать данную структуру в случае разработки микросервиса с простой логикой и минимумом взаимодействий со сторонними сервисами/базами данных и прочим. Так как в случае с гексагональной архитектурой нужно писать относительно много кода.

Для более детального ознакомления с гексагональной архитектурой рекомендую ознакомиться со статьей на habr , а также с докладом на GolangConf.

Монорепозитории

Под монорепозиторием будем понимать политику разработки, когда код нескольких проектов, библиотек располагается в одном репозитории с системой контроля версий. Крупные компании, такие как Google, Twitter, Facebook, Microsoft используют такой подход у себя.

Давайте рассмотрим плюсы и минусы монорепозиториев и попробуем понять, есть ли смысл применять такую стратегию.

При сервисной или микросервисной архитектуре часто приходится заботиться о версионности, сохранении обратной совместимости, необходимо думать об управлении зависимостями. В случае монорепозитория это менее выражено за счет того, что вся кодовая база находится в одном месте.

Сервис-ориентированная архитектура, в свою очередь, позволяет снизить связанность элементов системы, что помогает сделать каждый элемент изолированным, упростить совместную разработку. В монорепозитории выше вероятность возникновения мерж конфликтов, затрудняется логика релиза приложения. Также с ростом проекта сильно усложняется навигация и поиск нужной части кода.

В целом, концепция монорепозитория имеет место быть, особенно, если речь идет о новом или относительно небольшом проекте, потому что в этих случаях больше времени будет тратиться на поддержку нескольких репозиториев. Недостатком можно считать сильную связность кода, например, если мы поменяли логику работы кода в библиотеке, нам нужно убедиться, что она не поломает логику во всех местах,в которых используется. А это могут быть совершенно разные сервисы, про которые мы, как разработчики библиотеки, можем ничего не знать. Плюс ко всему, сложно работать с репозиторием, в котором тысячи или десятки тысяч файлов, как минимум потому, что нужно относительно много времени, чтобы спулить себе этот репозиторий. Более подробно о монорепозиториях можно почитать в статье на habr , статье на habr , статье на itnan.

Пример монорепозитория на Go monorepo-microservices .

Обзор project-layout

Данный шаблон можно найти в репозитории golang-standarts/project-layout.

Внимание! Несмотря на то, что в названии репозитория присутствует слово “standarts”, никакого отношения к стандартам Go и разработчикам языка не имеет.

В репозитории содержатся набор лучших практик, собранных сообществом, для структурирования крупных и средних проектов, а также некоторые комментарии и рекомендации. Например, сообщество рекомендует помещать внутрь папки проекта internal код, который вы не хотите, чтобы импортировали в свои библиотеки или сервисы. Папку pkg напротив рекомендуется использовать для размещения публичного кода. С этими и другими советами можно ознакомиться в репозитории.

Абстракции в Go

Паттерн “Accept interfaces return structs”

Для начала разберемся на примере в чем заключается суть данного паттерна

```
package repository


type FileRepo struct {
  fileName string
}


func NewFileRepo(fileName string) *FileRepo {
  return &FileRepo{fileName: fileName}
}


func (fr *FileRepo) Get(key string) (string, error) {
  // TODO: impl
  return "", nil
}


func (fr *FileRepo) Put(key, value string) error {
  // TODO: impl
  return nil
}
```

В представленном фрагменте кода мы видим функцию NewFileRepo, которая является “конструктором” для структуры FileRepo. Данная структура имеет два метода: Put и Get. Перед нами

стоит вопрос: а что же лучше возвращать: структуру FileRepo или же объявить интерфейс, который будет реализовывать структура FileRepo, и вернуть его, например такой:

```
type Repo interface {
  Get(key string) (string, error)
  Put(key, value string) error
}
```

Допустим, мы решили вернуть интерфейс:

```
func NewFileRepo(fileName string) Repo {
  return &FileRepo{fileName: fileName}
}
```

Однако, теперь, если мы решим добавить новый экспортируемый метод структуры FileRepo, нам придется вручную или с помощью дополнительных инструментов расширять наш интерфейс Repo. Плюс ко всему, мы можем не знать, что хочет видеть вызывающая сторона: структуру или же интерфейс. В случае, если мы вернем структуру, то “убьем двух зайцев”: сможем удовлетворить потребность вызывающей стороны и в структуре, и в интерфейсе(если, конечно, структура реализует данный интерфейс).

Предположим, что для работы некоторого сервиса необходим наш файловый репозиторий и мы его принимаем в “конструкторе” сервиса (функция New) как определенный нами выше в пакете repository интерфейс Repo:

```
package service

import (
  "log"

  web_broker "github.com/some-repo-example/web-broker/pkg/web-broker"
  "github.com/some-repo-example/web-broker/internal/app/repository"
)

type (
  Service struct {
```

```
    repo repository.Repo
  }
)

func New(repo repository.Repo) *Service {
  return &Service{repo: repo}
}

func (s *Service) Put(req *web_broker.PutValueReq) error {
  if err := s.repo.Put(req.Key, req.Value); err != nil {
    log.Printf("service/Put: put repo err: %v", err)
    return err
  }

  return nil
}

func (s *Service) Get(req *web_broker.GetValueReq) (*web_broker.GetValueResp,
error) {
  value, err := s.repo.Get(req.Key)
  if err != nil {
    log.Printf("service/Get: get from repo err: %v", err)
    return nil, err
  }

  return &web_broker.GetValueResp{Value: value}, nil
}
```

На первый взгляд все выглядит хорошо: принимаем интерфейс, возвращаем структуру. Но допустим мы захотели покрыть тестами наш сервис и написать для repo простейший мок:

```
...
type mockRepo struct {}
func (mockRepo) Get(key string) (string, error) {
  return "geekbrains", nil
}

func (mockRepo) Put(key, value string) error {
  return nil
}
...
func TestService(t *testing.T) {
  svc := New(mockRepo{})
  ...
}
```

Если при этом поменяется интерфейс Repo, допустим, добавится новый метод, мы уже не сможем в тесте использовать структуру mockRepo, наше приложение не соберется и нам придется в нашем

моке реализовывать этот новый метод, хотя он может быть совсем не нужен для нашего сервиса.

От этих проблем можно избавиться, если возвращать структуру при создании FileRepo:

```
func NewFileRepo(fileName string) *FileRepo {

  return &FileRepo{fileName: fileName}

}
```

И при этом изменить сигнатуру функции New при создании структуры сервиса следующим образом:

```
...

type (
  repo interface {
     Get(key string) (string, error)
     Put(key, value string) error
  }

  Service struct {
     repo repo
  }

)

func New(repo repo) *Service {
  return &Service{repo: repo}
}

...
```

Функция New принимает на вход интерфейс repo, определенный для этого же сервиса, и возвращает структуру Service. Это и есть простейшая реализация паттерна “Accept interfaces return structs”.

Также рабочим вариантом будет принятие структуры FileRepo в функции New пакета service:

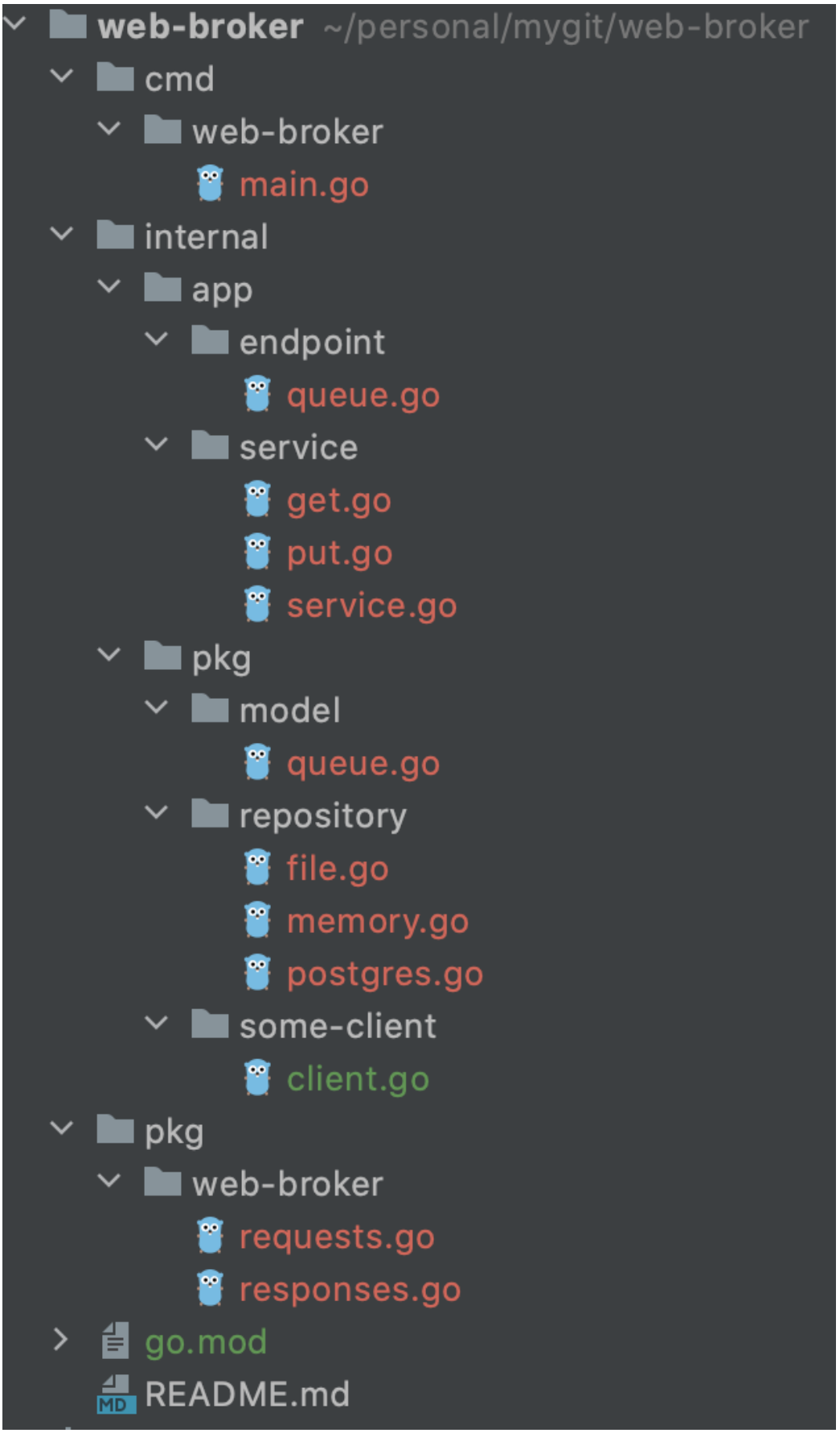
```
func New(repo repository.FileRepo) *Service {

  return &Service{repo: repo}
}
```

Но, во-первых, это затруднит тестирование вашего сервиса, во-вторых, доставит неудобства, если в дальнейшем вместо файлового репозитория вы захотите использовать, например, реляционную базу данных.

Внимание! Рассмотренный нами паттерн, как и любой другой, не может всегда подходить ко всем ситуациям. В большей части случаев выгоднее возвращать структуру, однако, есть случаи, когда оправдано возвращение интерфейса. Подробнее можно прочитать в статье.

Структура типового веб-сервиса

Так как одно из основных предназначений Go -- реализация веб-сервисов, давайте рассмотрим структуру типового веб-сервиса.


web-broker ~/personal/mygit/web-broker
cmd
web-broker
main.go
internal
app
endpoint
queue.go
service
get.go
put.go
service.go
pkg
model
queue.go
repository
file.go
memory.go
postgres.go
some-client
client.go
pkg
web-broker
requests.go
responses.go
go.mod
README.md

Перед нами структура небольшого проекта, в целом, он мог быть реализован в одном файле main.go, но для демо примера он был структурирован. В папке internal “традиционно” лежит код с внутренней логикой, в cmd лежит короткий main.go файл, который вызывает код из internal пакета для инициализации сущностей и запускает http сервер.

В internal/app лежит ключевая бизнес логика нашего сервиса в папке service, которой следует зависеть от абстракций (интерфейсов), в нашем случае это интерфейс repo. В пакете endpoint осуществляется маппинг запросов на методы нашего сервиса и преобразования результата в нужный нам формат(json и тп).

В internal/pkg располагаются вспомогательные пакеты для нашего сервиса, а именно пакет repository с заглушками реализации хранилища в виде файла, оперативной памяти, базы данных postgres. Все эти варианты реализуют интерфейс repo из пакета service, что позволяет бесшовно заменять их. Также для демонстрации был добавлен пакет некоторого клиента, который может быть использован нашим приложением.

В корневой папке pkg лежит код, который может быть полезен сторонним приложениям. Например, тут могут лежать сгенеренные protobuf’ом файлы, структуры запросов и ответов с проставленными json/xml/etc тегами для удобства клиентов данного сервиса, типы сущностей приложения и подобное.

Обратите внимание, что можно было всю бизнес-логику сервиса писать в файле service.go, однако, это может повлечь за собой некоторые проблемы, такие как:

- возникновение мерж конфликтов, если несколько разработчиков будут писать бизнес-логику в одном файле.

- нагромождение файла.в случае роста сервиса файл может занимать несколько тысяч, а то и десятков тысяч строк, что усложняет навигацию.

При разделении бизнес логики по файлам вероятность возникновения этих проблем ниже. Поэтому логика была разделена на файлы get.go и put.go. Не стесняйтесь разделять логику внутри пакета на файлы, если считаете, что это упростит навигацию, поддержку кода и разработку в командных условиях.

Отдельно стоит остановиться на названии ваших пакетов. Постарайтесь избегать пакетов с именованиями: “common”, “util” и подобных. Во-первых, они не всегда конкретно отражают свое предназначение, во-вторых, есть риск превращения этих пакетов в “свалку”, в-третьих, это потенциально место для возникновения циклических зависимостей.

Внимание! Вовсе необязательно, чтобы все ваши проекты были реализованы по данному шаблону. Структура не является стандартом, а лишь демонстрацией одного из типовых вариантов веб сервиса.

При создании структуры использовались рекомендации из project-layout, использовался паттерн “Accept interfaces, return structs”, “dependencies injection”(см доп материал 5). Детальнее можно ознакомиться с проектом по ссылке из доп материала 4.

Структура библиотеки

Рассмотрим, как можно применить best practices из project-layout для структурирования библиотеки и стоит ли это делать.

Для примера возьмем два кейса:

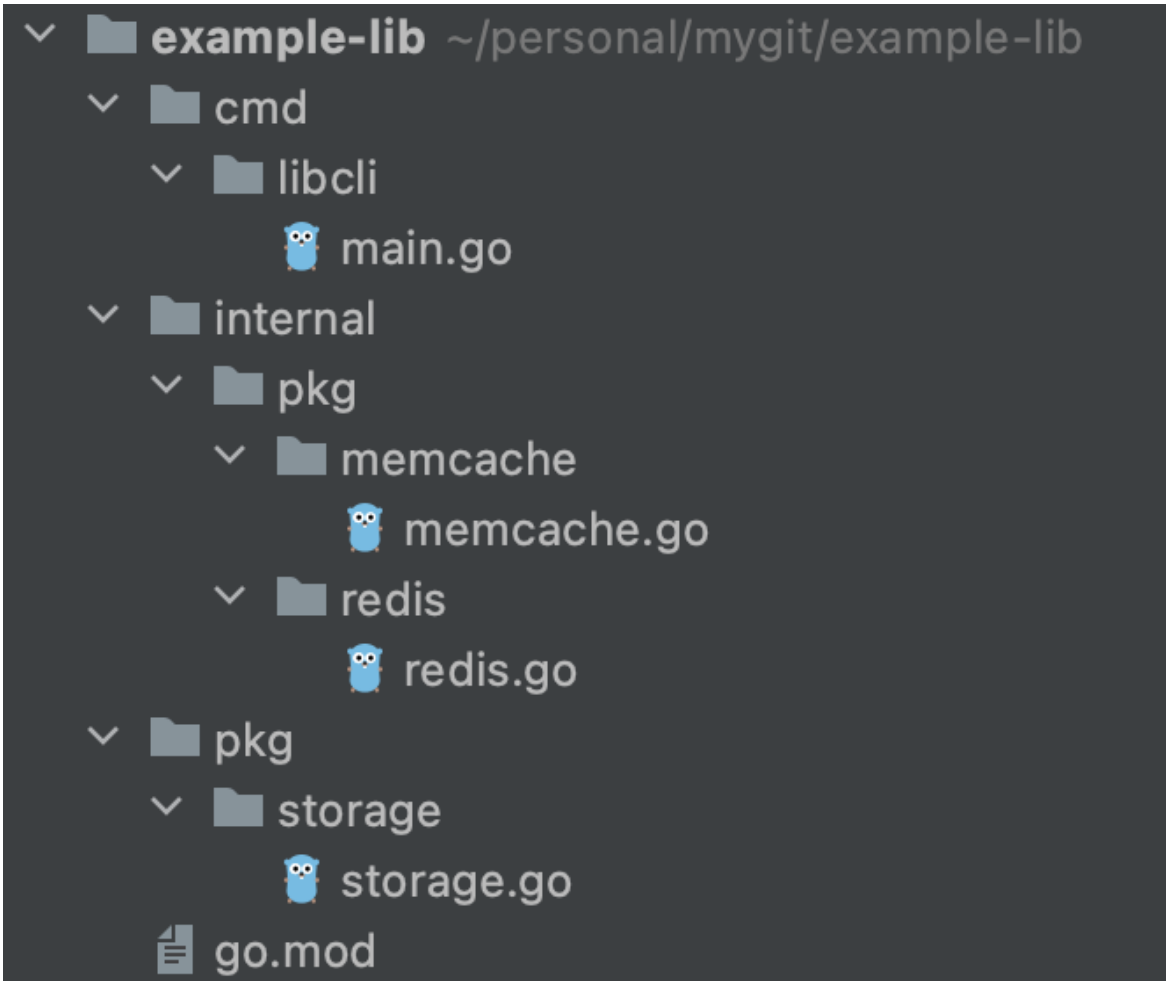
- Мы хотим написать небольшую библиотеку-клиент к какому-нибудь интернет ресурсу
- Мы хотим написать полноценную библиотеку для работы с kafka.

В первом случае нас скорее всего устроит плоская структура, так как логика небольшой библиотеки-клиента, вероятно, относительно простая и много кода вряд ли придется писать. Использование более сложного подхода к структуре в данном случае будет неоправданным.

Рассмотрим второй кейс. Здесь логика может быть сложнее, и на первый взгляд может показаться, что для реализации такой библиотеки стоит прибегнуть к более сложной структуре. Однако, если посмотреть в репозитории популярных гошных библиотек для работы с kafka, а именно segmentio и sarama , их структура похожа на плоскую, нет ярковыраженного структурирования на внутреннюю и внешнюю логику.

Если же мы бы хотели применить рекомендации из project-layout, то структура библиотеки могла бы

выглядеть следующим образом:



Начнем с cmd каталога. Если в случае веб сервиса здесь располагалась входная точка, то в случае библиотеки здесь могут располагаться cli инструменты, которые связаны с нашей библиотекой. Из названий файлов и пакетов понятно, что эта библиотека работает с in-memory хранилищам данных, в таком случае в cmd может лежать написанная нами утилита командной строки для работы с хранилищем. В вашей библиотеки может вовсе не быть этого функционала, поэтому в части случаев можно будет обойтись и без cmd каталога.

В internal каталоге, как и в случае с типовым веб-сервисом, лежит код, с внутренней логикой библиотеки, в нашем случае это реализация функционала для работы с memcache и redis.

Если опираться на project-layout, то в pkg каталоге следует размещать публичный код, который предназначен для клиентов нашей библиотеки.

Следует отметить, что для значительной части библиотек может быть вполне оправдана плоская структура, потому что часто библиотеки предназначены для решения узкой проблемы и масштабного расширения не планируется.

Практическая часть

Рассмотрим на примере типового веб-сервиса, что именно лежит внутри пакетов и как реализована работа с абстракциями.

Начнем с пакета service, при инициализации структуры сервиса мы делаем ее зависимой от абстракции репозитория, что позволит использовать любую реализацию, удовлетворяющую данному интерфейсу

```
1  package service
2
3  import (
4          "github.com/vlslav/web-broker/internal/pkg/model"
5  )
6
7  type (
8          repo interface {
9                  Get(key string) (string, error)
10                 Put(putReq *model.PutValue) error
11         }
12
13         Service struct {
14                 repo repo
15         }
16 )
17
18 func New(repo repo) *Service {
19         return &Service{repo: repo}
20 }
```

Для простоты чтения, навигации и для удобного внесения совместных правок методы нашего сервиса разнесены по разным файлам,таким образом service.go отвечает за саму структуру сервиса и ее инициализацию, а два файла get.go и put.go за получение и размещение данных соответственно. За счет зависимости бизнес логики от интерфейсов мы можем замокать внешние зависимости сервиса и легко протестировать логику.

main web-broker / internal / app / service /

Vladislav Lyashenko init		
..		
get.go		init
put.go		init
service.go		init

Пакет endpoint отвечает за маппинг роутов на конкретные методы нашего сервиса, а также за парсинг ответа и маршаллинг ответа во что бы то ни было: json, xml, etc

```go
package endpoint

import (
	"net/http"

	"github.com/gorilla/mux"
	web_broker "github.com/vlslav/web-broker/pkg/web-broker"
)

type queueSvc interface {
	Put(req *web_broker.PutValueReq) error
	Get(req *web_broker.GetValueReq) (*web_broker.GetValueResp, error)
}

func RegisterPublicHTTP(queueSvc queueSvc) *mux.Router {
	r := mux.NewRouter()

	r.HandleFunc("/{queue}", putToQueue(queueSvc)).Methods(http.MethodPut)
	r.HandleFunc("/{queue}", getFromQueue(queueSvc)).Methods(http.MethodGet)

	return r
}

func putToQueue(queueSvc queueSvc) http.HandlerFunc {
	return func(w http.ResponseWriter, request *http.Request) {
		// TODO: parse req and call queueSvc.Put(...)
	}
}

func getFromQueue(queueSvc queueSvc) http.HandlerFunc {
	return func(w http.ResponseWriter, request *http.Request) {
		// TODO: parse req and call queueSvc.Get(...)
	}
}
```

В корневом pkg пакете, как уже упоминалось, лежат структуры запросов в наш сервис, которые могут быть полезны клиентам нашего сервиса

Также рядом располагается файл со структурой ответов.

main web-broker / pkg / web-broker / requests.go /

Vladislav Lyashenko init

0 contributors

12 lines (10 sloc) | 151 Bytes

```
1  package web_broker
2
3  type (
4          GetValueReq struct {
5                  Key string
6          }
7
8          PutValueReq struct {
9                  Key   string `json:"key"`
10                 Value string `json:"value"`
11         }
12 )
```

В internal/pkg располагаются пакеты, необходимые внутренней логике нашего приложения, например, реализация работы с хранилищем:

```
package repository

import "github.com/vlslav/web-broker/internal/pkg/model"

type MemRepo struct{}

func NewMemRepo() *MemRepo {
        return &MemRepo{}
}

func (mr *MemRepo) Get(getReq string) (string, error) {
        // TODO: impl
        return "", nil
}

func (mr *MemRepo) Put(putReq *model.PutValue) error {
        // TODO: impl
        return nil
}
```

Также, если, например, наше приложение взаимодействует с другими сервисы, мы можем расположить пакет клиента здесь

Глоссарий

Практическое задание

1. Изменить структуру типового веб-сервиса так, чтобы любое хранилище можно было создавать через New, а не через NewPgRepo и тд.
2. Укажите, какие могут быть проблемы в структуре пакетов repository и endpoint из кода типового веб-сервиса, а также предложить решения по снижению вероятности возникновения этих проблем и реализовать их.

Дополнительные материалы

1. https://github.com/katzien/go-structure-examples
2. https://github.com/AlexanderGrom/go-patterns

3. https://habr.com/ru/post/267125/ + https://golangconf.ru/2020/abstracts/6954
4. https://github.com/vlslav/web-broker
5. Про dependency injection https://www.youtube.com/watch?v=pTwtcP5QgII + https://habr.com/ru/company/funcorp/blog/372199/

Используемые источники

1. https://www.youtube.com/watch?v=oL6JBUk6tj0 GopherCon 2018: Kat Zien - How Do You Structure Your Go Apps
2. https://mycodesmells.com/post/accept-interfaces-return-struct-in-go
3. https://golangconf.ru/2020/abstracts/6954