О спикере

- Работаю в ООО "РОБОВОЙС" на должности Senior Developer
- Занимаюсь интеграциями средств связи
- 6 лет опыта разработки
- 4 года опыта в Golang

Цель интенсива

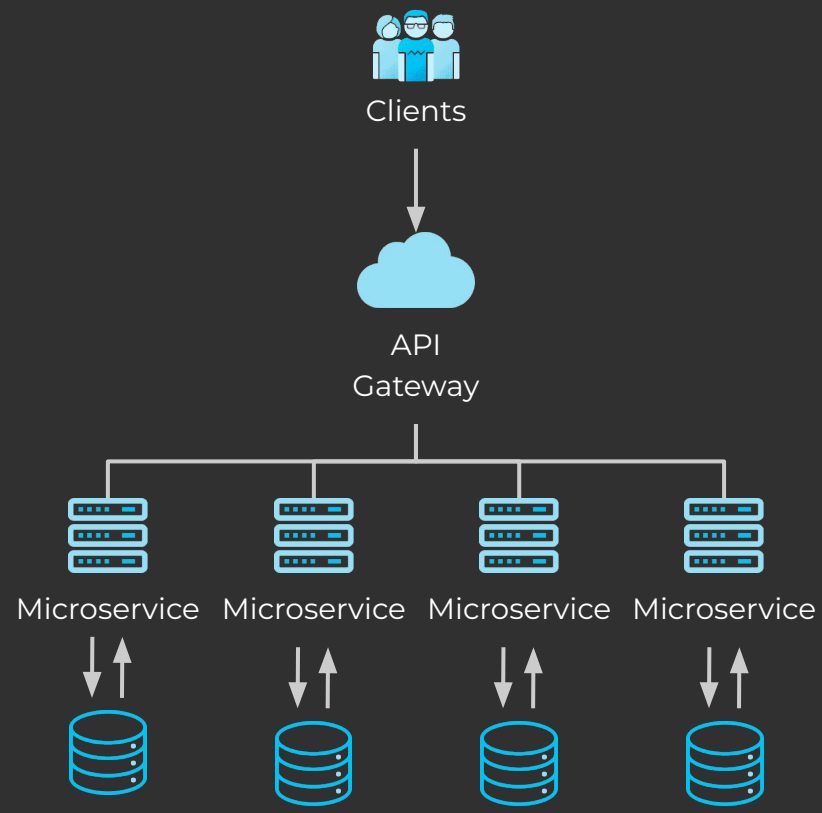
Что вы получите:

- Навыки работы в чистой архитектуре
- Навыки вывода логов
- Навыки работы с трассировкой
- Пример написания тестов

Что будет будет сделано:

- Полноценный микросервис с REST API (контакт сервис)
- Документация к REST API
- Логирование
- Трассировка
- Тесты

Все практические задания связаны между собой

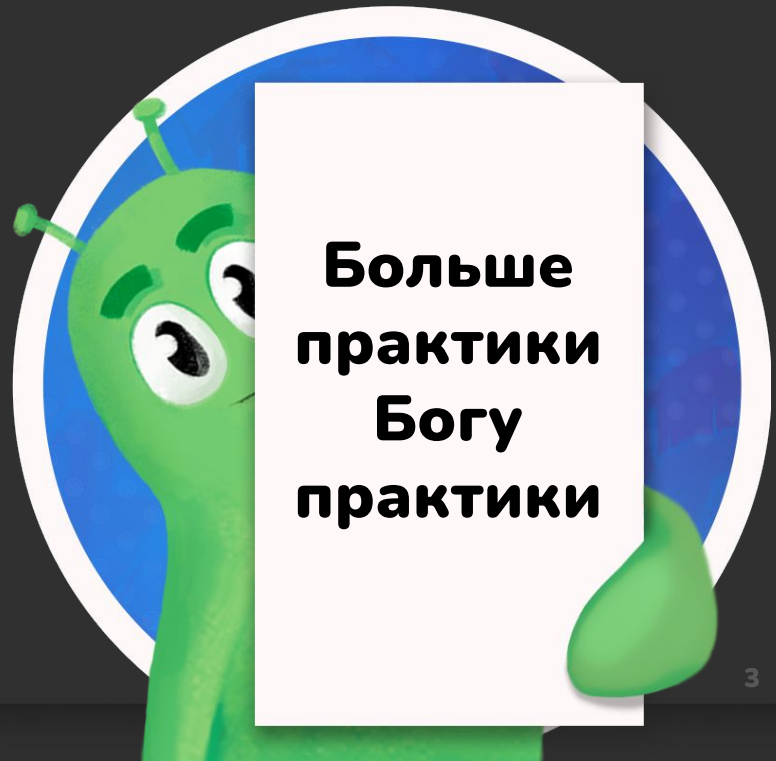


{REST API}

Вводная

Как будет проходить теория?

- Презентация, нацеленная на выполнение практики
- Минимальная информация для выполнение практики

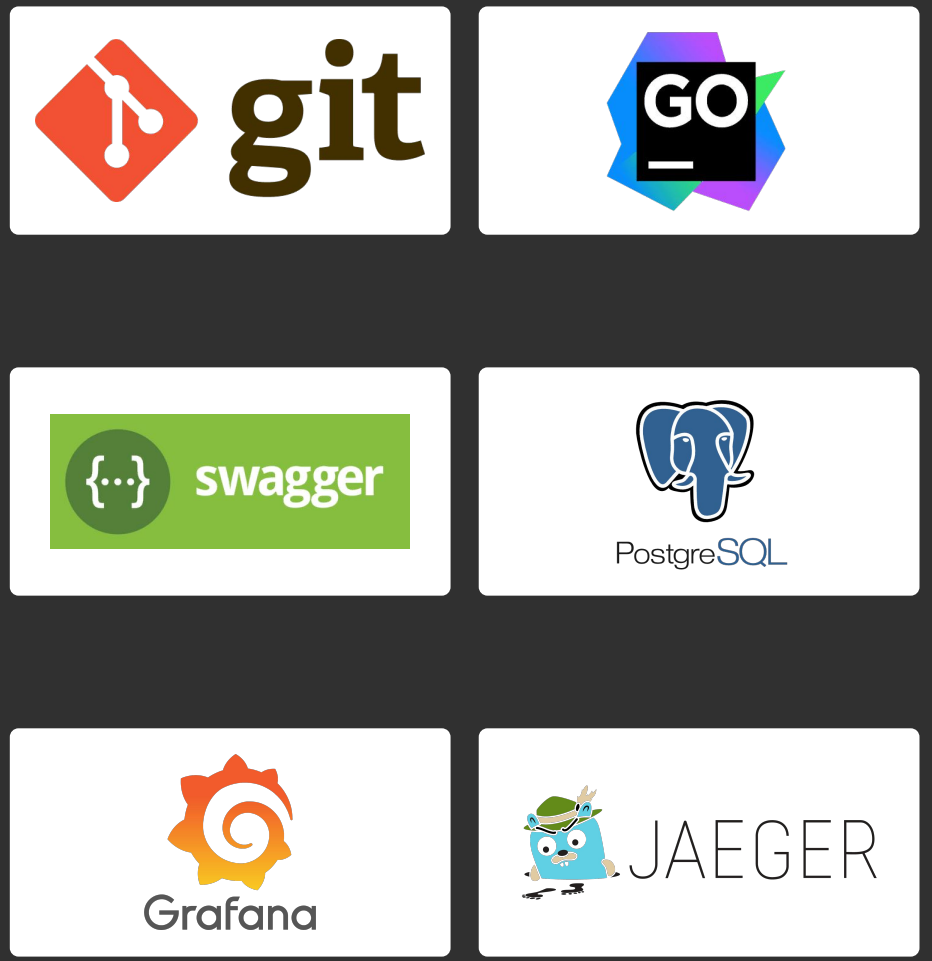


Стек технологий

Как будет проходить практика?

- **Требуются:**
  - IDE Goland
  - Swagger
  - GIT
- **Доступы к:**
  - GIT
  - Grafana
  - Jaeger
  - PostgreSQL

Все практические задания связаны между собой. Каждое новое практическое задание расширяет возможности предыдущего.



Таков путь...

День 1:

1. Создание структуры проекта

День 2:

2. Подключение к БД
3. Чистая архитектура:
   - 3.1. Структура папок по чистой архитектуре
   - 3.2. Наполнение – Domain
   - 3.3. Интерфейс – Use Case
   - 3.4. Интерфейс – Repository
   - 3.5. Конструкторы слоёв
   - 3.6. Инициализация слоёв на main
   - 3.7. Реализация интерфейса – Use Case

День 3:

   - 3.8. Реализация интерфейса – Repository
   - 3.9. Реализация – Delivery
4. Использование – Context
5. Добавить логи
6. Добавить трассировку
7. Тестирование**

Что будет в курсе?

Структура проекта
(теория и практика)

Observability
(теория и практика)

Чистая архитектура
(теория и практика)

Тестирование
(демонстрация)

3 Дня

8 Часов теории

16 Часов практики

Правила

Где задавать вопросы?

1. В чате Telegram

Где получать ответы?

1. В конце каждого дня или смыслового блока
2. В чате Telegram в течение прохождения интенсива

Как будут выбираться вопросы?

Будут выбраны самые интересные или часто задаваемые вопросы,

ставьте реакции.

Структура проекта

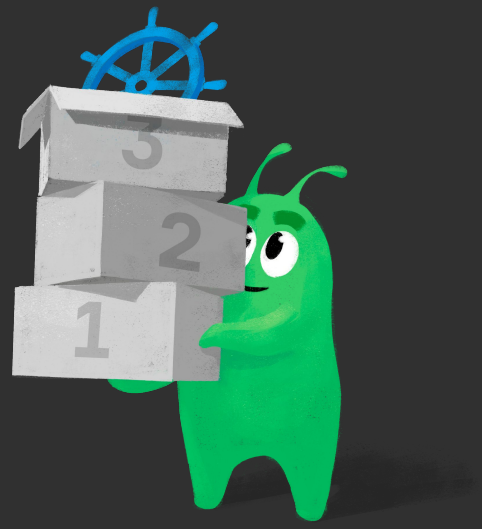
Зачем нужна структура проекта?

Разбор основных папок /
Просмотр примеров проектов на Go

Иерархия папок

Зачем нужна

структура проекта?



Зачем нужна структура проекта

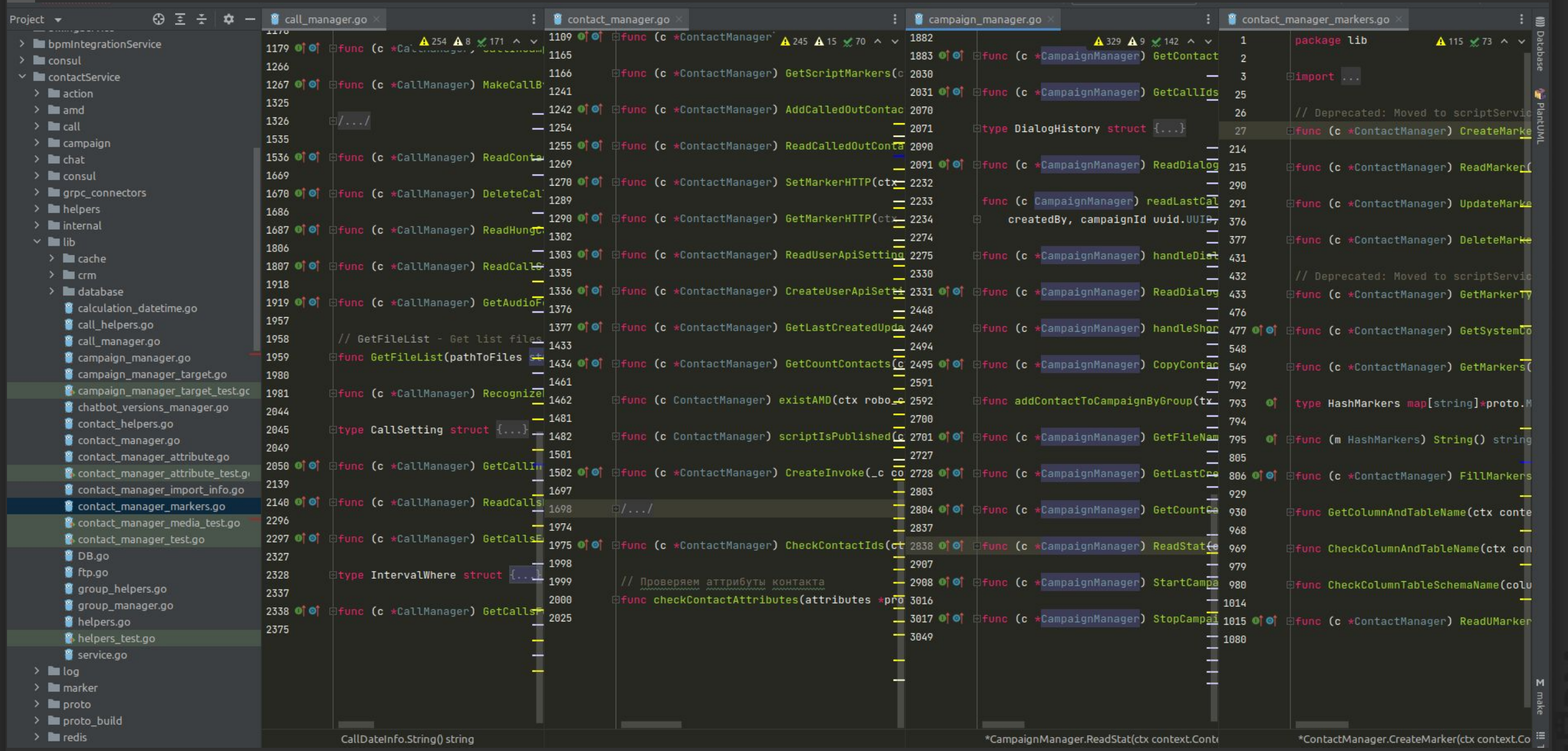
1. Зачем нужна структура проекта?

2. Придерживаетесь ли вы шаблонов при создании новых проектов?

3. Есть ли у вас сложности при работе в вашей структуре проекта?

Что если:

делать, не думая о структуре?


Project
bpmIntegrationService
consul
contactService
action
amd
call
campaign
chat
consul
grpc_connectors
helpers
internal
lib
cache
crm
database
calculation_datetime.go
call_helpers.go
call_manager.go
campaign_manager.go
campaign_manager_target.go
campaign_manager_target_test.go
chatbot_versions_manager.go
contact_helpers.go
contact_manager.go
contact_manager_attribute.go
contact_manager_attribute_test.go
contact_manager_import_info.go
contact_manager_markers.go
contact_manager_media_test.go
contact_manager_test.go
DB.go
ftp.go
group_helpers.go
group_manager.go
helpers.go
helpers_test.go
service.go
log
marker
proto
proto_build
redis
call_manager.go
contact_manager.go
campaign_manager.go
contact_manager_markers.go
func (c *CallManager) MakeCallB
func (c *CallManager) ReadConta
func (c *CallManager) DeleteCal
func (c *CallManager) ReadHungC
func (c *CallManager) ReadCalls
func (c *CallManager) GetAudioF
// GetFileList - Get list files
func GetFileList(pathToFiles
func (c *CallManager) Recognize
type CallSetting struct {...}
func (c *CallManager) GetCallI
func (c *CallManager) ReadCalls
func (c *CallManager) GetCallsF
type IntervalWhere struct {...}
func (c *CallManager) GetCallsF
func (c *ContactManager) GetScriptMarkers(c
func (c *ContactManager) AddCalledOutContac
func (c *ContactManager) ReadCalledOutConta
func (c *ContactManager) SetMarkerHTTP(ctx
func (c *ContactManager) GetMarkerHTTP(ctx
func (c *ContactManager) ReadUserApiSetting
func (c *ContactManager) CreateUserApiSetti
func (c *ContactManager) GetLastCreatedUpda
func (c *ContactManager) GetCountContacts(c
func (c ContactManager) existAMD(ctx robo_c
func (c ContactManager) scriptIsPublished(c
func (c *ContactManager) CreateInvoke(_c co
/.../
func (c *ContactManager) CheckContactIds(ct
// Проверяем аттрибуты контакта
func checkContactAttributes(attributes *pro
func (c *CampaignManager) GetContact
func (c *CampaignManager) GetCallIds
type DialogHistory struct {...}
func (c *CampaignManager) ReadDialog
func (c CampaignManager) readLastCal
createdBy, campaignId uuid.UUID,
func (c *CampaignManager) handleDist
func (c *CampaignManager) ReadDialog
func (c *CampaignManager) handleShor
func (c *CampaignManager) CopyContac
func addContactToCampaignByGroup(tx
func (c *CampaignManager) GetFileNam
func (c *CampaignManager) GetLastCre
func (c *CampaignManager) GetCounte
func (c *CampaignManager) ReadState
func (c *CampaignManager) StartCampa
func (c *CampaignManager) StopCampai
package lib
import ...
// Deprecated: Moved to scriptServic
func (c *ContactManager) CreateMarke
func (c *ContactManager) ReadMarker(
func (c *ContactManager) UpdateMarke
func (c *ContactManager) DeleteMarke
// Deprecated: Moved to scriptServic
func (c *ContactManager) GetMarkerTy
func (c *ContactManager) GetSystemCo
func (c *ContactManager) GetMarkers(
type HashMarkers map[string]*proto.M
func (m HashMarkers) String() string
func (c *ContactManager) FillMarkers
func GetColumnAndTableName(ctx conte
func CheckColumnAndTableName(ctx con
func CheckColumnTableSchemaName(colu
func (c *ContactManager) ReadUMarker
CallDateInfo.String() string
*CampaignManager.ReadStat(ctx context.Conte
*ContactManager.CreateMarker(ctx context.Co

contact.proto

```
1925
1926
1927    service ActionService {...}
1941
1942    service ContactService {...}
2006
2007    service GroupService {...}
2039
2040    service CampaignService {...}
2106
2107     // TODO: Delete if it's dead code
2108    service ChatbotVersionService {...}
2124
2125    service ScriptService {...}
2128
2129    service CallService {...}
2182
2183    service AMDService {...}
2190
2191    service DialogService {...}
2200
2201     // ChatTest Service
2202    service ChatTest {...}
2208
2209    service Crm {...}
2212
```

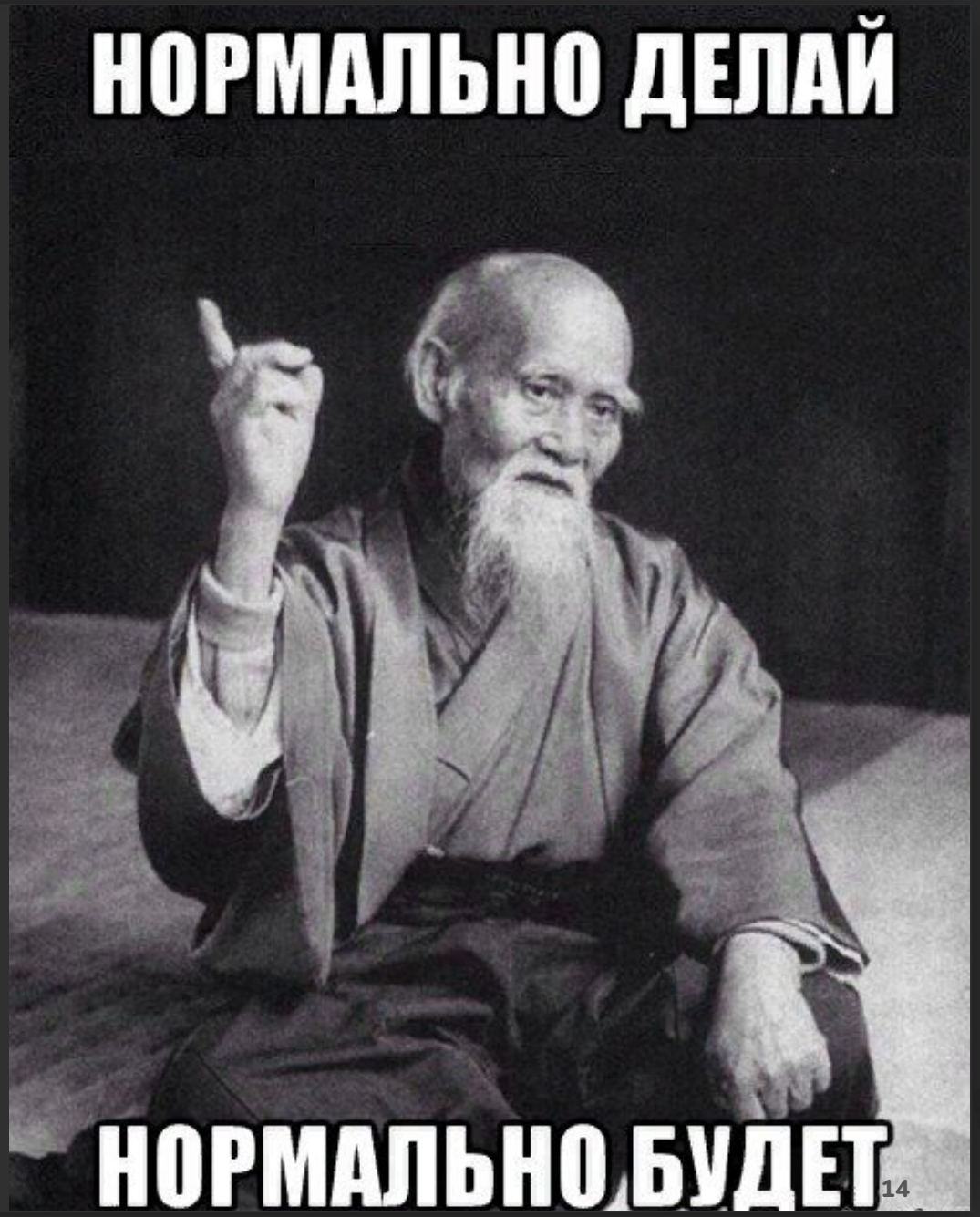
Демонстрация плохого кода

(Ветка: 001_bad_code)

А как нужно ?

Цели структуры проекта:

1. Ускорение разработки
2. Отражение целей проекта



Разбор основных папок

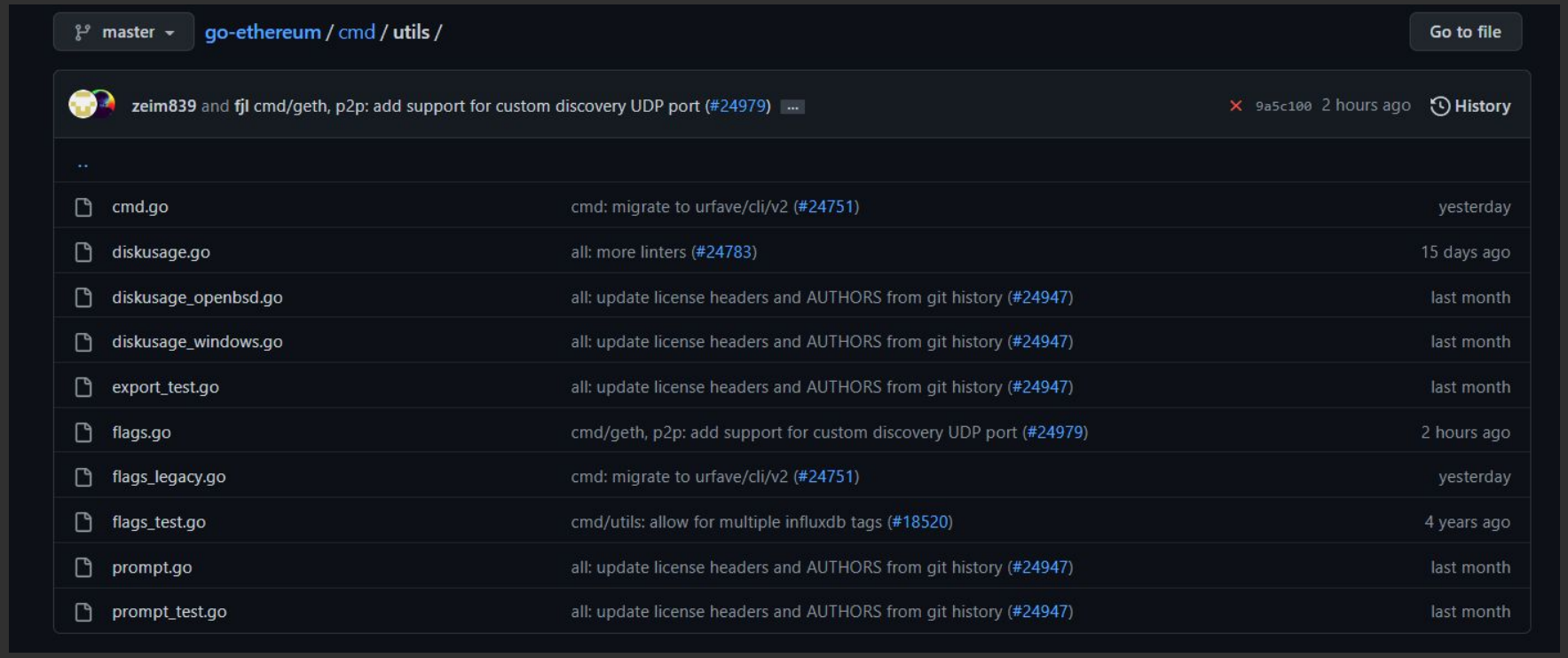
Разбор основных папок

- **cmd** – Основные приложения проекта
- **internal** – Внутренний код приложения и библиотек
- **pkg** – Код библиотек
- **vendor** – Зависимости приложений
- **docs / api –** Документация
- **deployments / deploy –** Шаблоны и файлы конфигураций
- **configs** – Шаблоны файлов конфигураций и файлы настроек по-умолчанию

Дополнительную информацию можно получить в [github.com]

cmd – Основные приложения проекта (git)

- Малые объёмы кода
- cmd/app – основное приложение
- cmd/... – вспомогательные приложения



internal – Внутренний код приложения и библиотек

Это код, который не должен быть применен в других приложениях и библиотеках (git).

Стоит отметить, что этот шаблон навязан самим компилятором Golang.

Ознакомьтесь с release notes Go 1.4

master viper / internal / encoding / Go to file

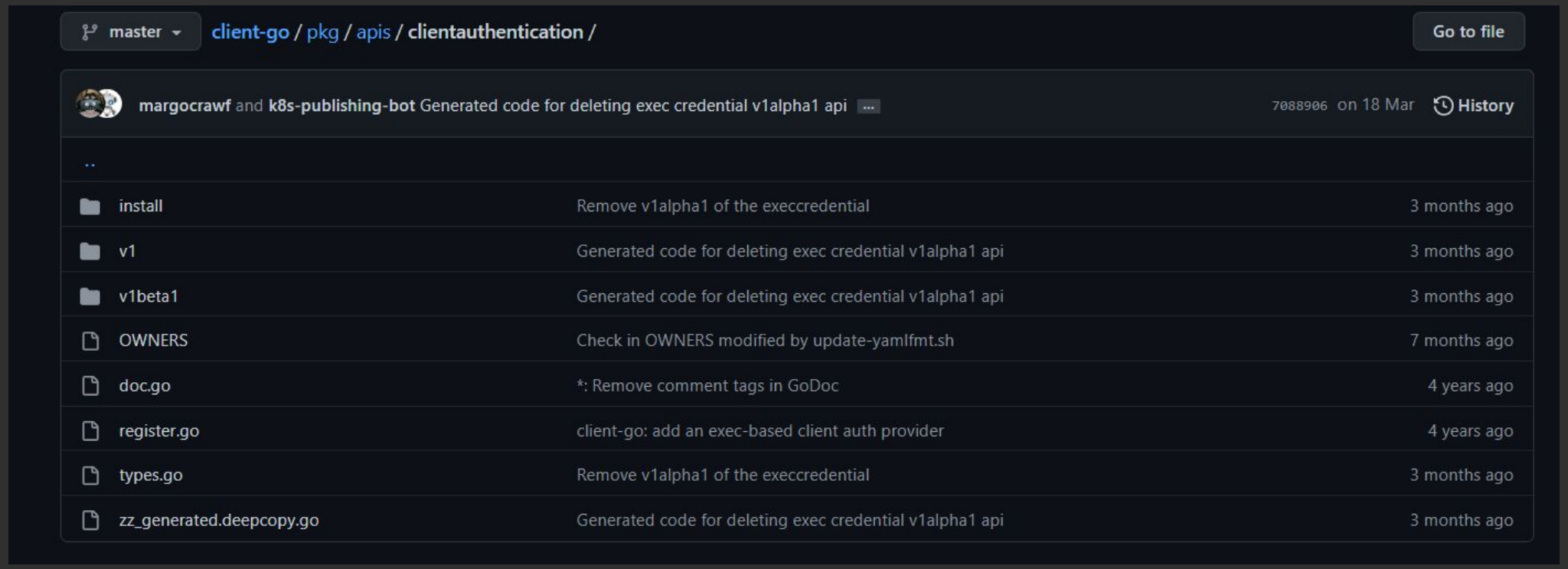
sagikazarmark feat: make toml2 the default ... 4322cf2 on 26 May History

..		
dotenv	feat(encoding): add dotenv codec	7 months ago
hcl	test(encoding): add tests for existing encoding implementations	7 months ago
ini	feat(encoding): add ini codec	7 months ago
javaproperties	feat(encoding): integrate Java properties codec into Viper	7 months ago
json	test(encoding): add tests for existing encoding implementations	7 months ago
toml	feat: make toml2 the default	last month
yaml	feat: make yaml3 the default	last month
decoder.go	feat(encoding)!: accept a map in the decoder interface	7 months ago
decoder_test.go	feat(encoding)!: accept a map in the decoder interface	7 months ago
encoder.go	feat(encoding)!: accept a map in the encoder interface	7 months ago
encoder_test.go	feat(encoding)!: accept a map in the encoder interface	7 months ago
error.go	feat(encoding): Encoder/Decoder registry implementations	12 months ago

pkg – Код библиотек

Это код, который пригодный для использования в сторонних приложениях (git)

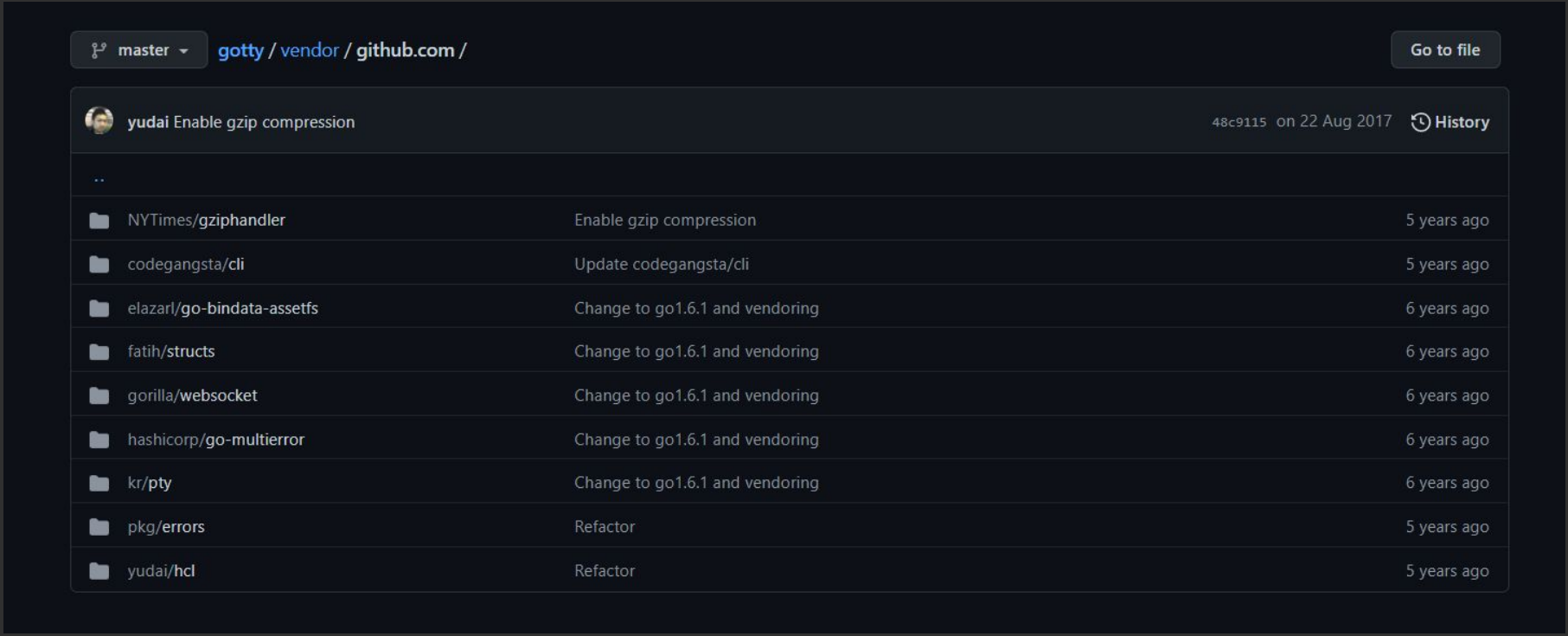
- Автономная работа
- Код в этой директории могут безопасно использовать другие



vendor – Зависимости приложений

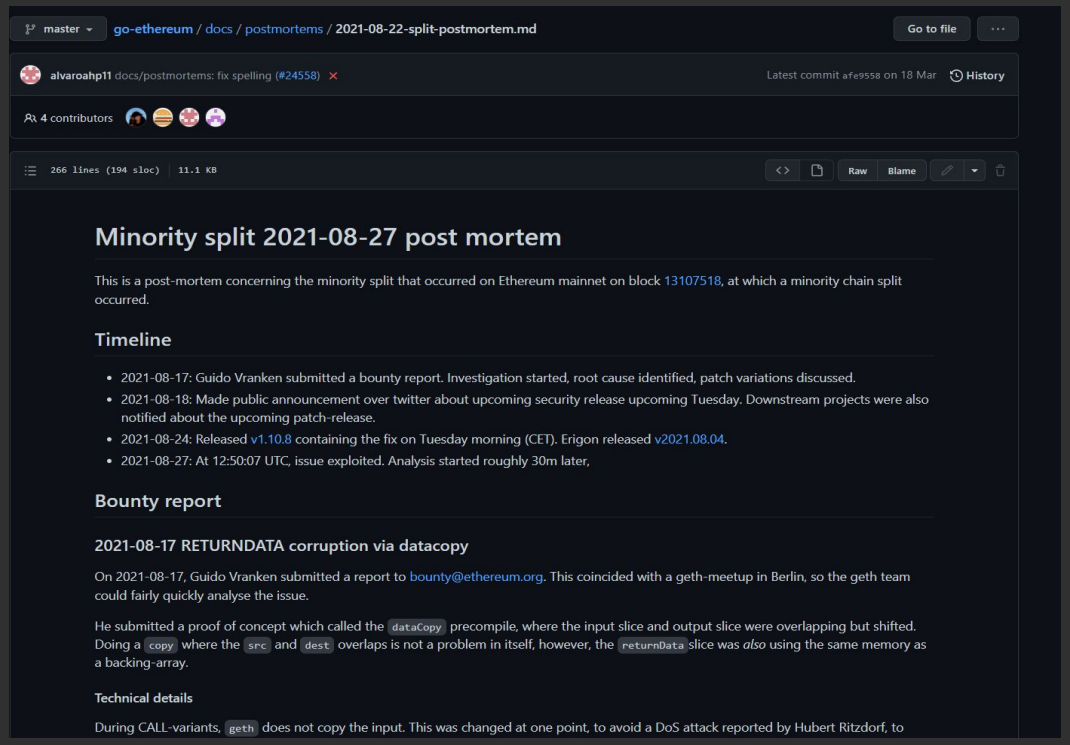
Управляемый вручную или с использованием (git)

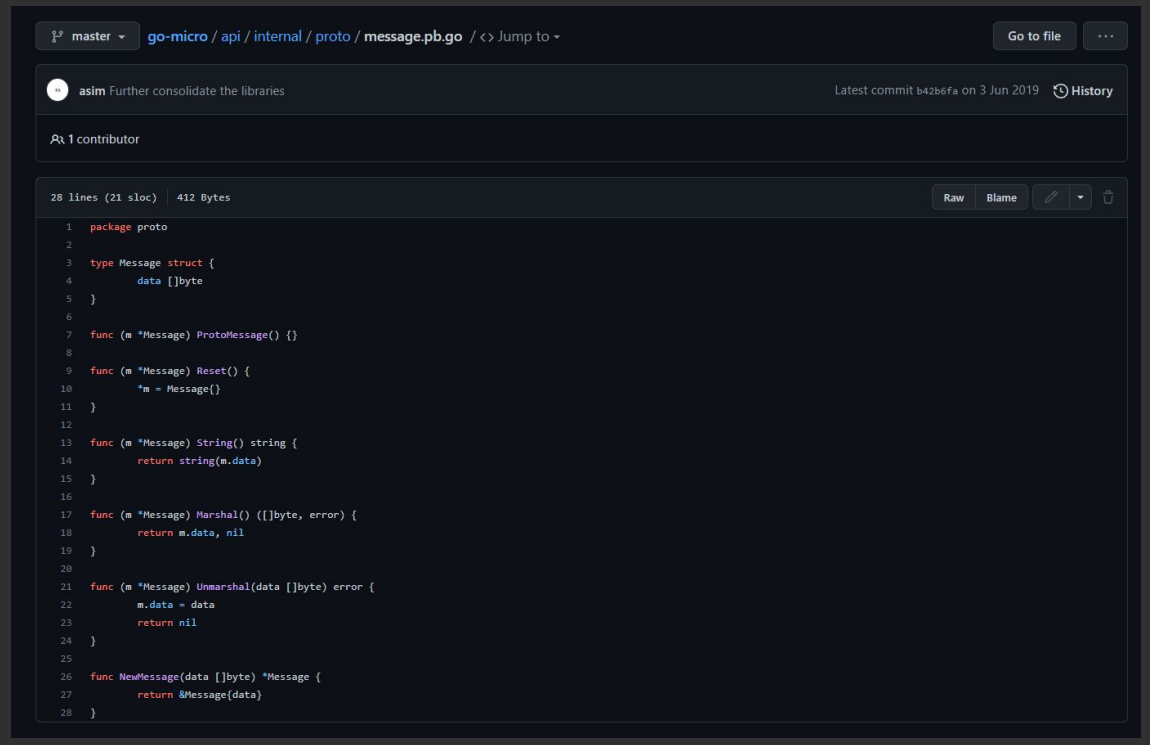
Например, использовать go mod



docs / api – Документация

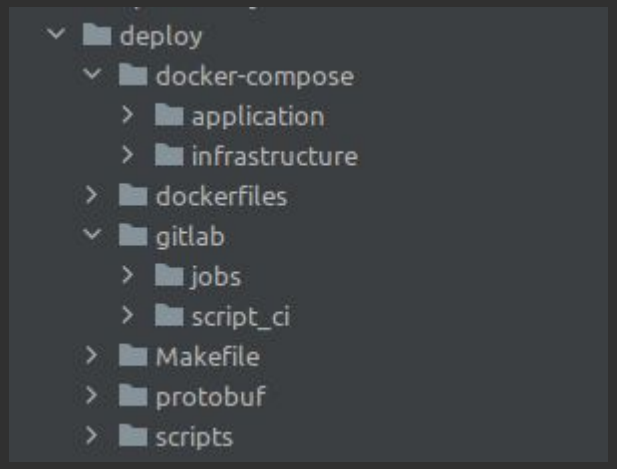
- **docs –** Проектные и пользовательские документы (git)
- **api** – Спецификации OpenAPI/Swagger, файлы схем JSON, файлы определения протоколов. (git)





deployments / deploy – Шаблоны и файлы конфигураций

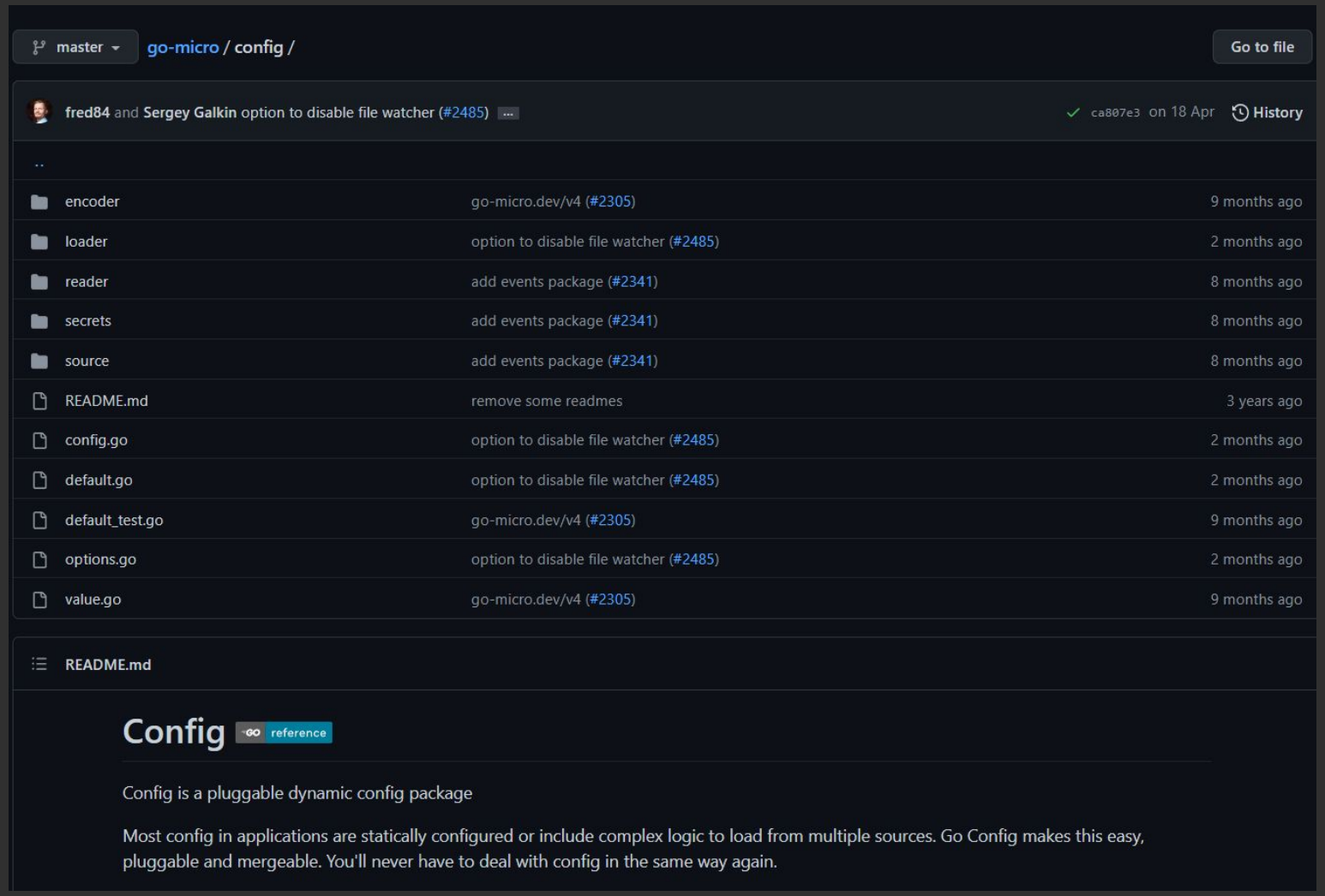
(docker-compose, kubernetes/helm, mesos, terraform, bosh)



service.yml

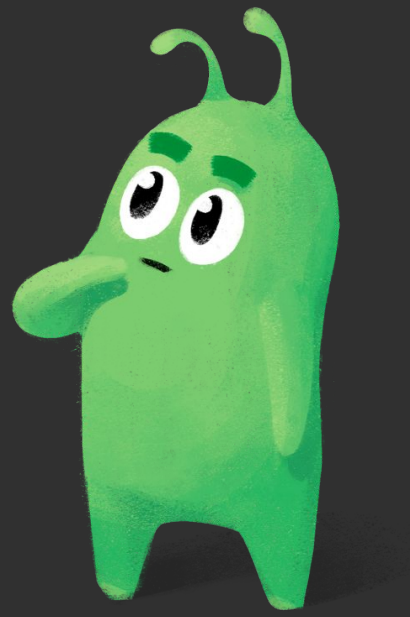
```yaml
version: "3.7"

networks:
  net:
    driver: overlay

services:
  contact:
    image: ${tag_contactservice}
    deploy:
      mode: replicated
      placement: <1 key>
      replicas: ${REPLICA}
      resources:
        limits:
          cpus: "0.25"
      restart_policy: <3 keys>
      update_config: <3 keys>
      labels: <13 items>
    volumes: <4 items>
    extra_hosts:
      - "speechcat:${SPEECHCAT_IP}"
    healthcheck: <5 keys>
    environment: <7 keys>
    logging: <2 keys>
    networks:
      - net
```

configs – Шаблоны файлов конфигураций и файлы настроек по-умолчанию (git)



Ответьте на вопросы

- *Как у вас организована структура?*
- *Проблемы при работе с структурой проекта?*
- *Есть вопросы по папкам?*
- *Есть шаблон при создании нового проекта?*

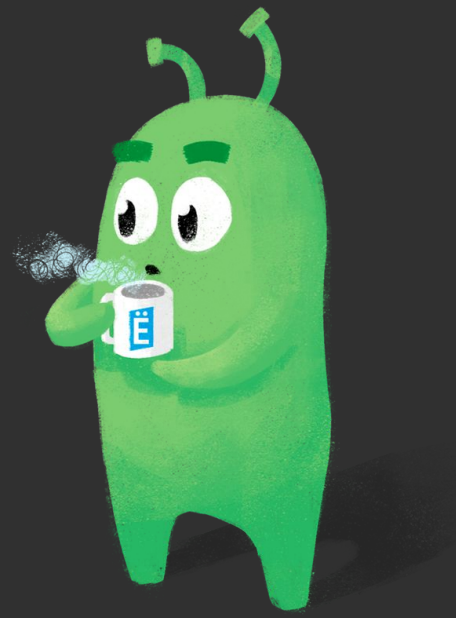


Итоги

Что делать если у вас особый случай?

- Нет подходящей папки
- Нет иерархии
- Много ненужных папок
- И вообще, это сложно....

Перерыв 10 мин



Иерархия папок

- project
  - pkg
  - services
    - serviceName
      - internal
        - cmd/app
        - configs
        - migrations
  - vendor

Практика: Создание структуры проекта (40 мин)

Цель: Подключиться к git и создать базовый проект для дальнейшей работы с ним

Задание:

1. Подключиться к git Слёрм
   a. Подключиться по адресу https://gitlab.slurm.io/a.iskakov/forgoproject
   b. Доступы для git можно получить в личном кабинете Слёрм
   c. Создайте свою ветку в git. Правила наименования ФамилияИнициалы-01
      Например, kolyadkons-01

2. Создать примитивную структуру проекта
   a. Создать папку pkg
   b. Создать папку services/contact/cmd
      i. Создать пакет app
      ii. Создать main функцию с выводом в консоль

3. Запустить проект

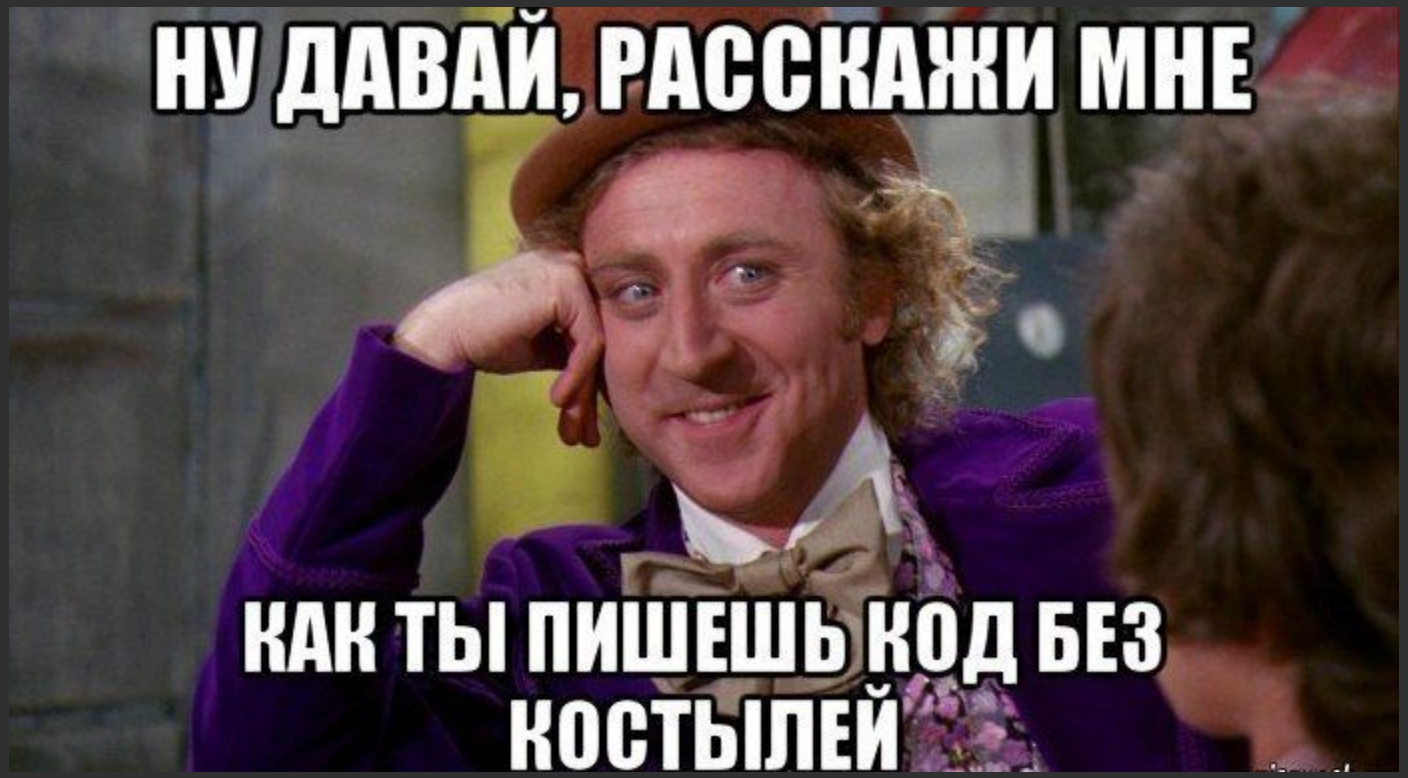
4. Залейте свою ветку в git

Результаты

Успели ли выполнить задание? (опрос)

- ❑ Да
- ❑ Успел на 75%
- ❑ Успел на 50%
- ❑ Успел на 25%
- ❑ Нет

Демонстрация

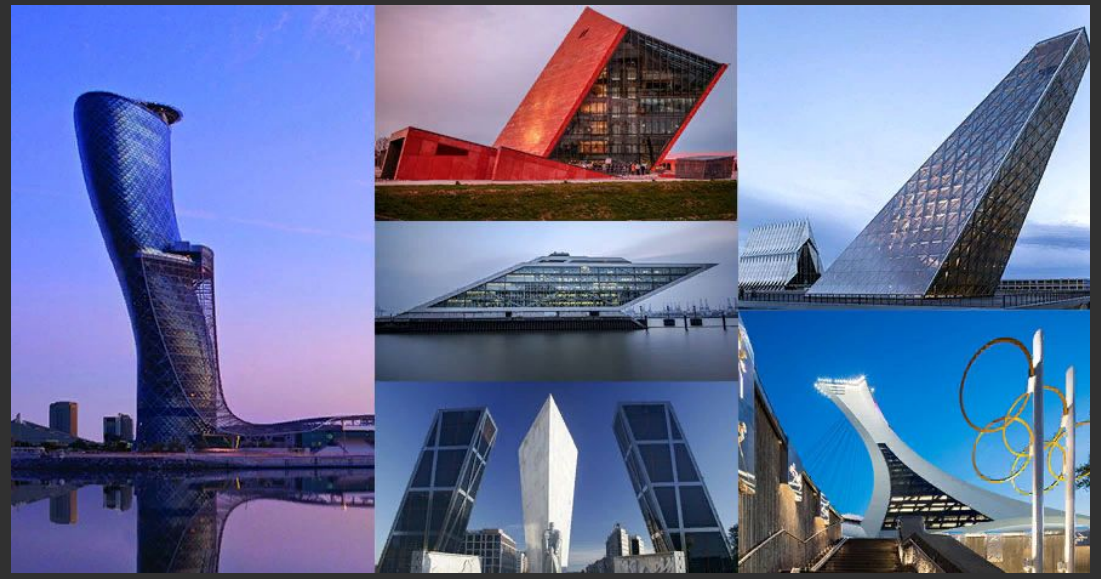


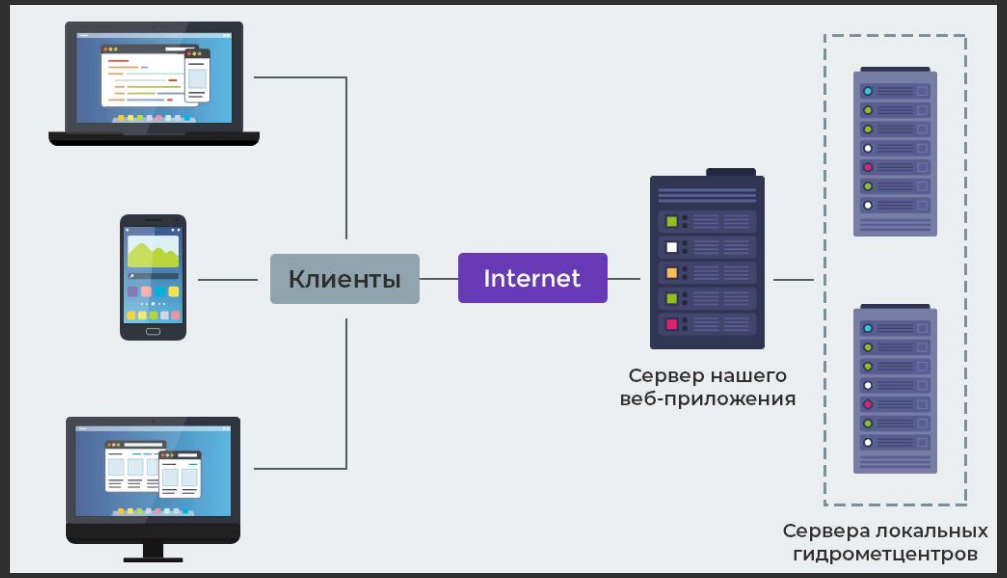
Обед 60 мин

Чистая архитектура

- Что такое архитектура?
- Что такое чистая архитектура?
- Зачем её использовать?
- Слои чистой архитектуры
- Правило зависимостей
- Пересечение границ

Что такое архитектура?





Архитектура - Компромисс

Блиц-опрос

1. Как хорошо/быстро ориентируетесь в структуре нового проекта?
2. Как быстро можете понять, что написано в коде?
3. Практикуете написание тестов?
4. Есть сложности с покрытиями конкретных функций тестами?
5. Боитесь вносить изменения в код, т.к. не понимаете, что может сломаться?

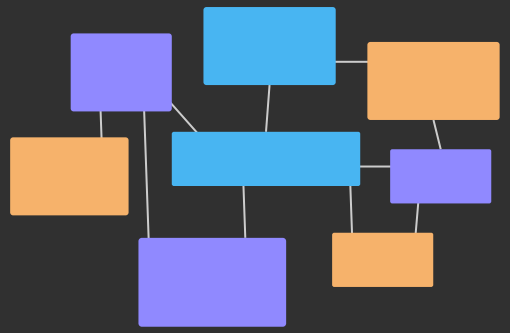
Что объединяет эти вопросы?

Low coupling, high cohesion

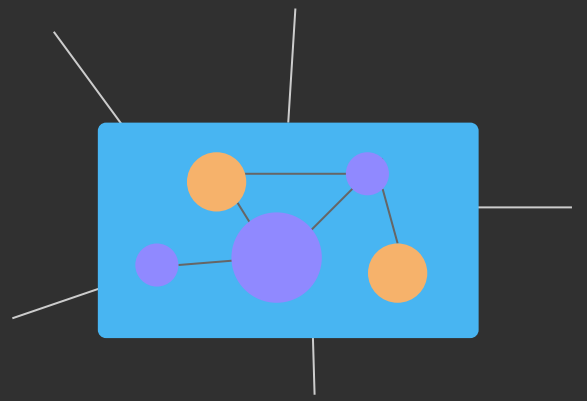
Качественный дизайн архитектуры обладает:

- **Low Coupling** – слабое связывание
- **High Cohesion** – высокое сцепления

Это значит, что программный компонент имеет мало внешних связей и отвечает за решение близких по смыслу задач



Coupling



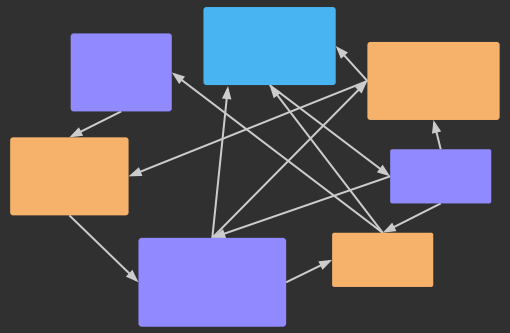
Cohesion

Coupling – Связанность

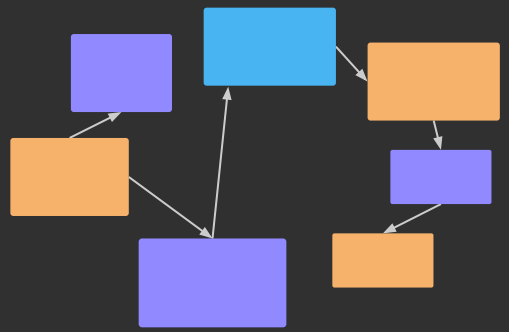
Coupling – насколько взаимозависимы разные подпрограммы или модули

- Low coupling – соответствует общим показателям хорошей читаемости и сопровождаемости
  - Малое число зависимостей между классами / подсистемами
  - Слабая зависимость одного класса / подсистемы от изменений в другом классе / подсистемы
  - Высокая степень повторного использования подсистем

- High coupling – затрудняет понимание логики модулей, их модификацию, автономное тестирование, а также переиспользование по отдельности



High Coupling



Low Coupling

Coupling – Связанность

- **Content – содержимого**
  - Модуль изменяет / полагается на внутренние особенности другого модуля
  - Изменение работы второго модуля приведет к переписыванию первого
- **Common – через общее**
  - Два модуля работают с общими данными (глобальной переменной)
  - Изменение разделяемого ресурса приведет к изменению всех работающих с ним модулей
- **External – через внешнее**
  - Два модуля используют навязанный извне формат данных (протокол связи)
- **Control – по управлению**
  - Один модуль управляет поведением другого
  - Присутствует передача информации о том, что и как делать

Coupling – Связанность

- **Data-structured – структурированным по данным**
  - Модули используют одну и ту же структуру, но каждый использует только ее части
  - Изменение структуры может привести к изменению модуля

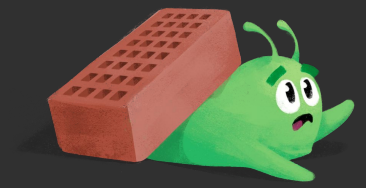
- **Data – через данных**
  - Модули совместно используют данные, например, через параметры

- **Message – по сообщениям**
  - Модули общаются только через передачу параметров или сообщений

- **No coupling – отсутствие связанности**
  - Модули вообще никак не взаимодействуют

Coupling – Связанность

Типы моделей связанности



Содержимого

Через общее

Через внешнее

По управлению

Структурированным по данным

Через данных

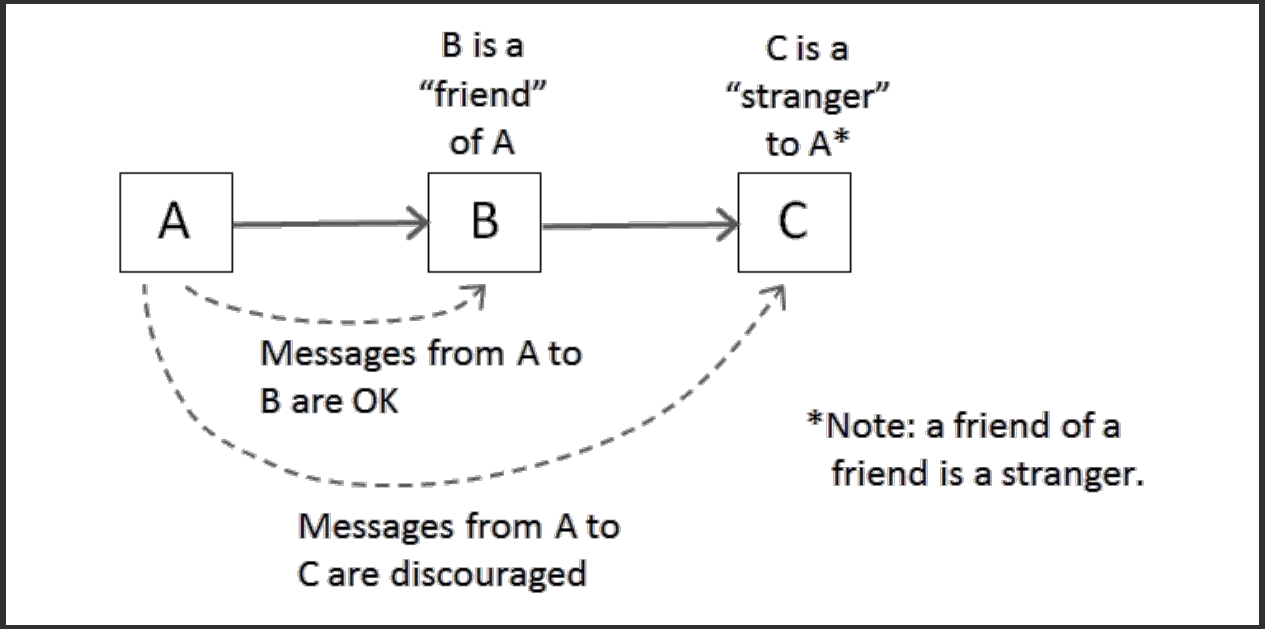
По сообщениям

Отсутствие связанности

Принцип наименьшего знания (Закон Деметры)

Объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B и у объекта B есть доступ к объекту C

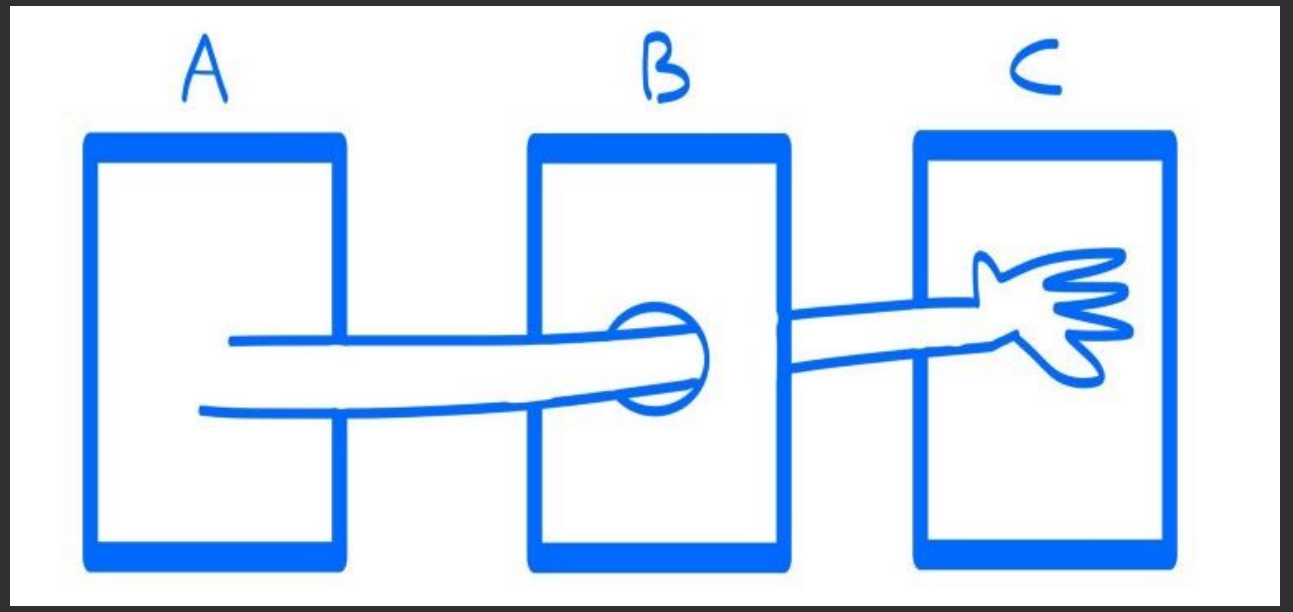
- Должен знать о модулях, которые «непосредственно» относятся к этому модулю
- Обращаться только к непосредственным «друзьям»
- Не взаимодействовать с незнакомцами



Принцип наименьшего знания (Закон Деметры)

Преимущества – код, разработанный с соблюдением данного закона, делает написание тестов более простым, а разработанное ПО менее сложно при поддержке и имеет большие возможности повторного использования кода

Недостатки – требуется создание большого количества малых методов-адаптеров для передачи вызовов метода к внутренним компонентам



Cohesion – Сцепление

Cohesion – насколько сильно взаимосвязаны элементов внутри модуля

- Low cohesion – много разнородных функций или несвязанных между собой обязанностей
  - Трудность понимания
  - Сложность при повторном использовании
  - Сложность поддержки
  - Ненадежность, постоянная подверженность изменениям

- High cohesion – обязанности хорошо согласованы между собой и он не выполняет огромных объемов работы

Cohesion – Сцепление

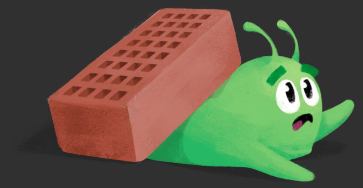
- **Coincidental – случайная**
  - Части модуля сгруппированы “от фонаря”
  - Единственное, что их объединяет — сам модуль
- **Logical – логическая**
  - Части модуля логически относятся к одной проблеме
  - Части могут различаться по своей природе
- **Temporal – временная**
  - Части модуля обычно используются в программе в одно время, рядом
- **Procedural – процедурная**
  - Части модуля всегда используются в определенном порядке

Cohesion – Сцепление

- **Communication – по взаимодействию**
  - Части модуля работают над одним и теми же данными
- **Sequential – по последовательности действий**
  - Результат работы одной части модуля является исходными данными для другой
- **Functional – функциональная**
  - Части модуля направлены на решение одной четкой задачи, за которую отвечает модуль

Cohesion – Сцепление

Типы моделей сцепления



Случайная

Логическая

Временная

Процедурная

По взаимодействию

По последовательности действий

Функциональная

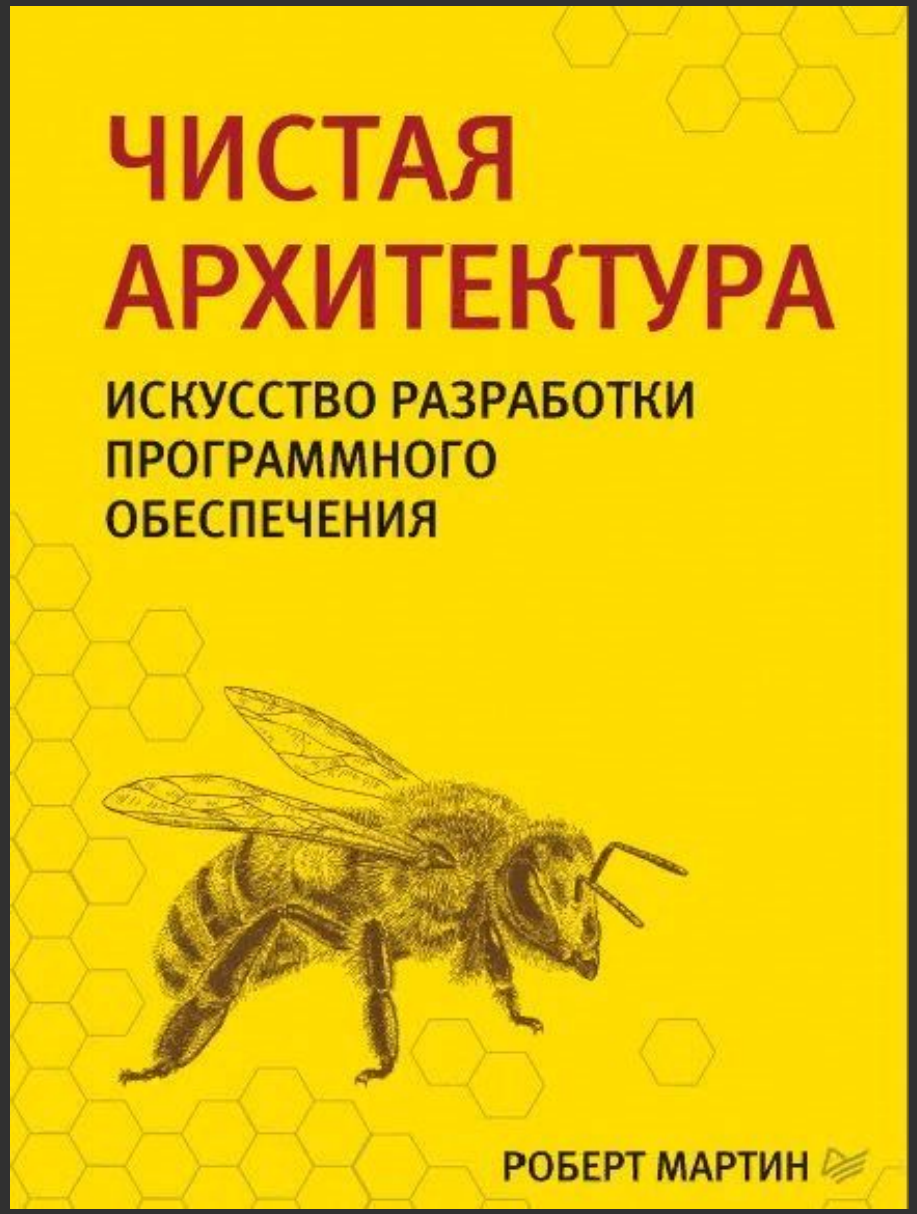
Что такое чистая архитектура?

Способ организации кода, который **способствует** строгому **разделению ответственности**

При этом между ними передаются только те ресурсы, которые необходимы для выполнения поставленной задачи

Способ разделения логики кода на части (слои):

- бизнес-правила
- обращение к внешним системам
- получение внешних запросов (API)



Зачем использовать чистую архитектуру?

Стандартизация проектов / модулей / микросервисов

Поддержка старых проектов

Быстрое переключение разработчиков между проектами

Независимость от сторонних систем

Быстрый старт нового проекта

Тестирование системы.
Скорость и удобство написания тестов

Зачем использовать чистую архитектуру?

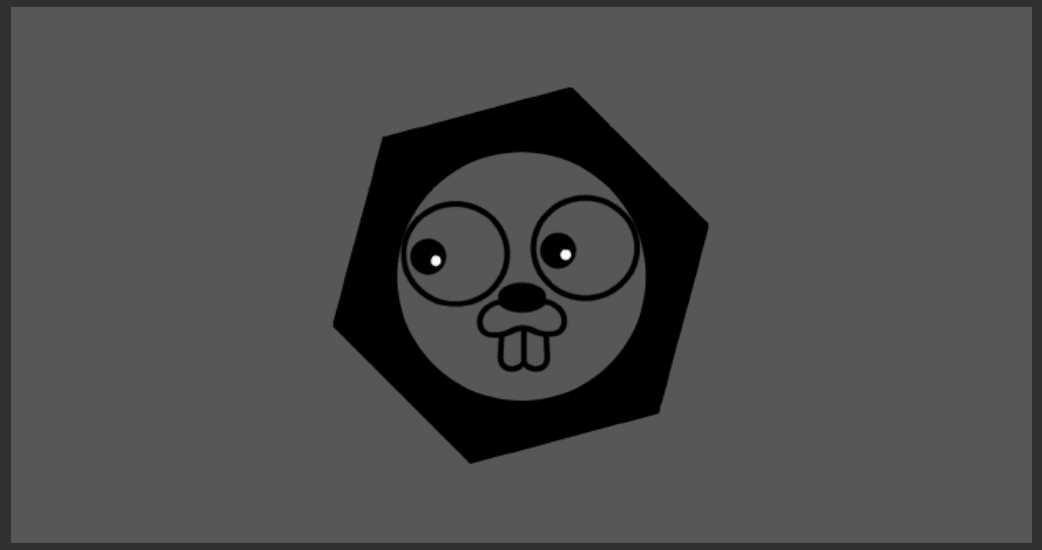
Независимость от сторонних систем:

- Фреймворков
- UI
- Базы данных
- Внешних сервисов

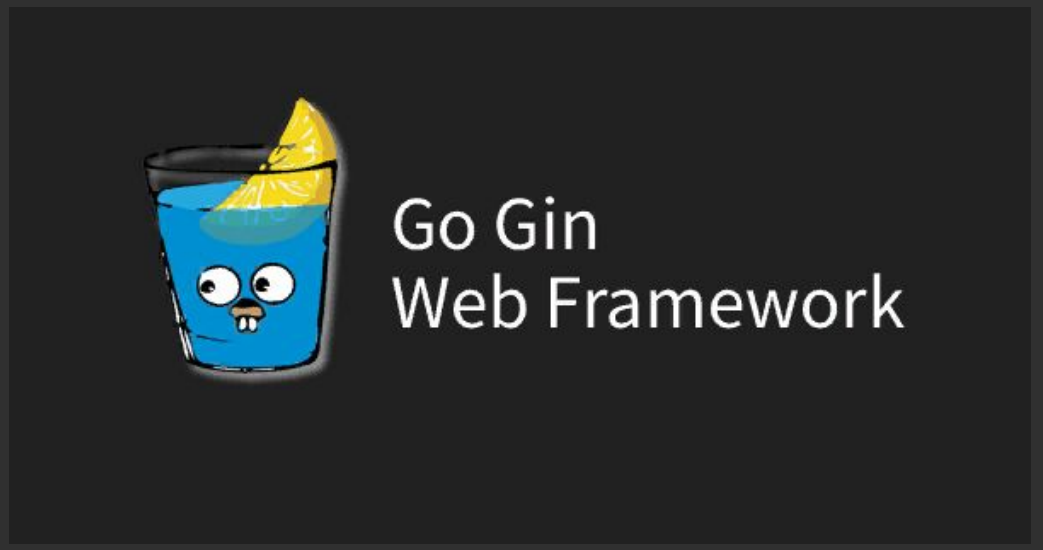
Независимость от сторонних систем

Фреймворк – программная платформа

Kit – набор пакетов, которые обеспечивают комплексный, надежный и создания микросервисов для организаций любого размера



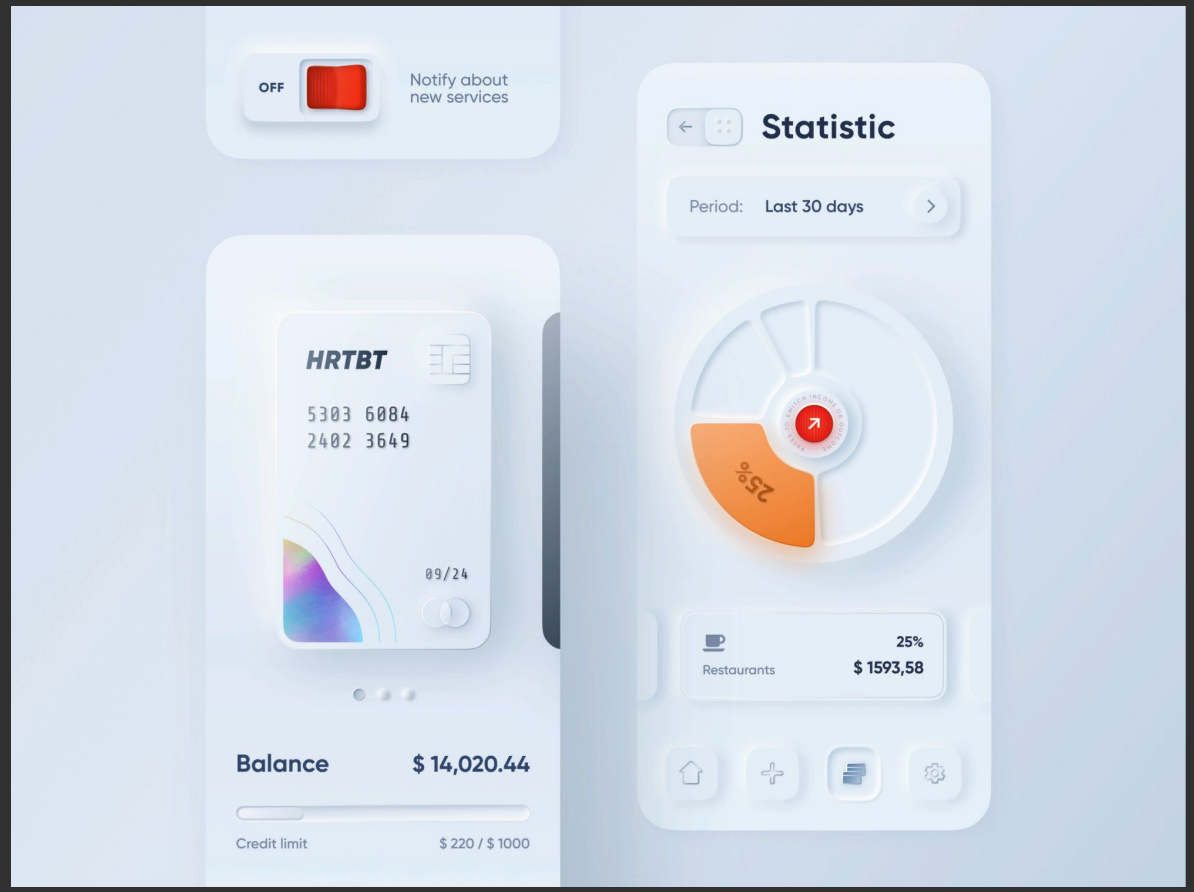
Gin – REST API для серверной части



Независимость от сторонних систем

UI – элементы продукта, делающих его притягательным:

- цвет
- стиль кнопок
- графика
- анимация
- типография
- инфографика
- виджеты
- поведение
- отклики кнопок
- и так далее



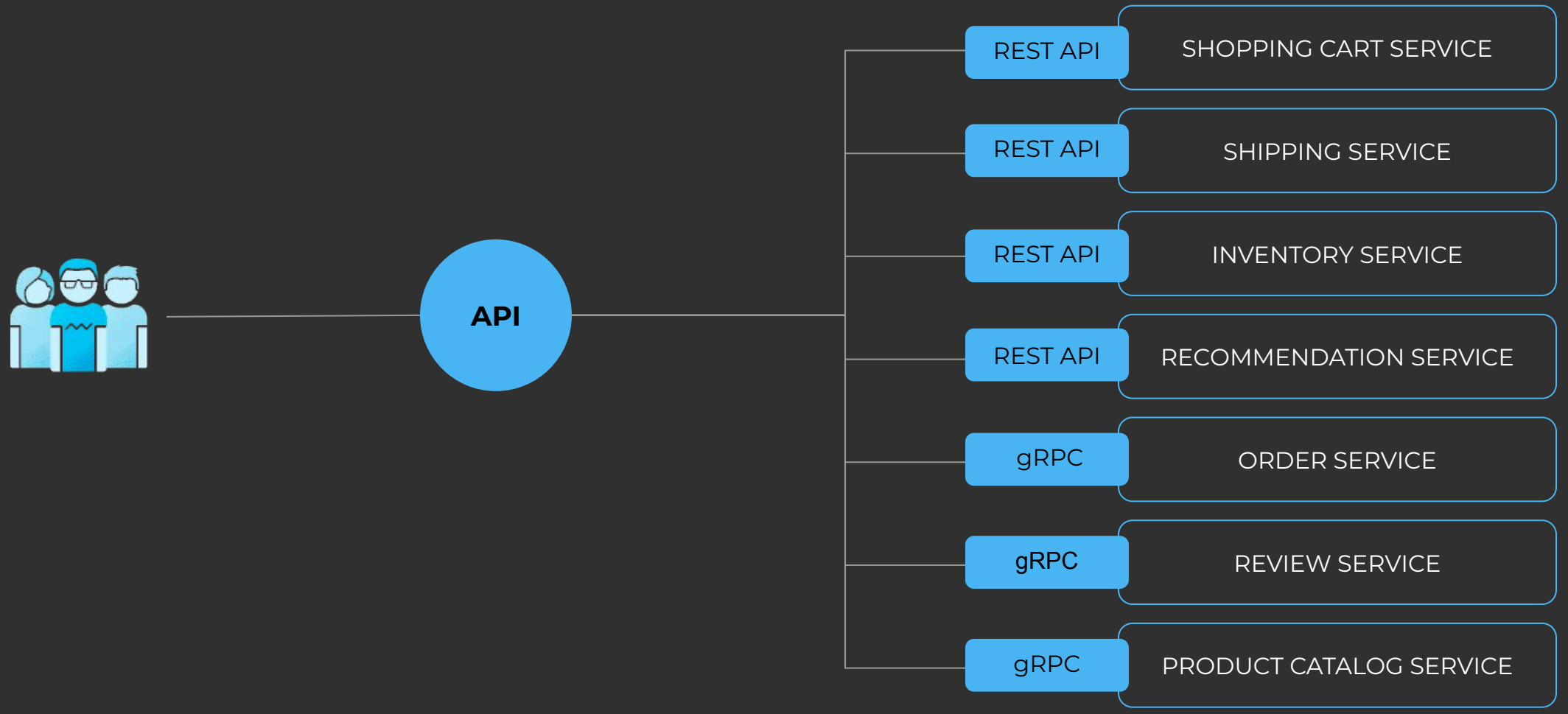
Независимость от сторонних систем

Базы данных – совокупность данных, хранимых в соответствии со схемой данных

- PostgreSQL
- ClickHouse
- MySQL
- Redis

Независимость от сторонних систем

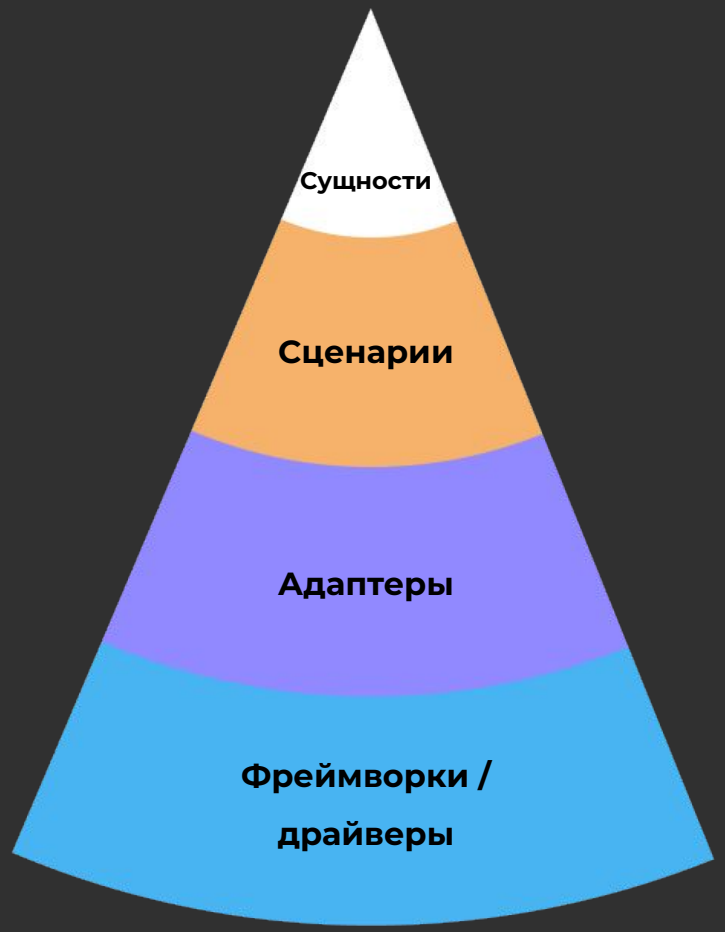
Внешних сервисов – любые сторонние системы



Перерыв 10 мин

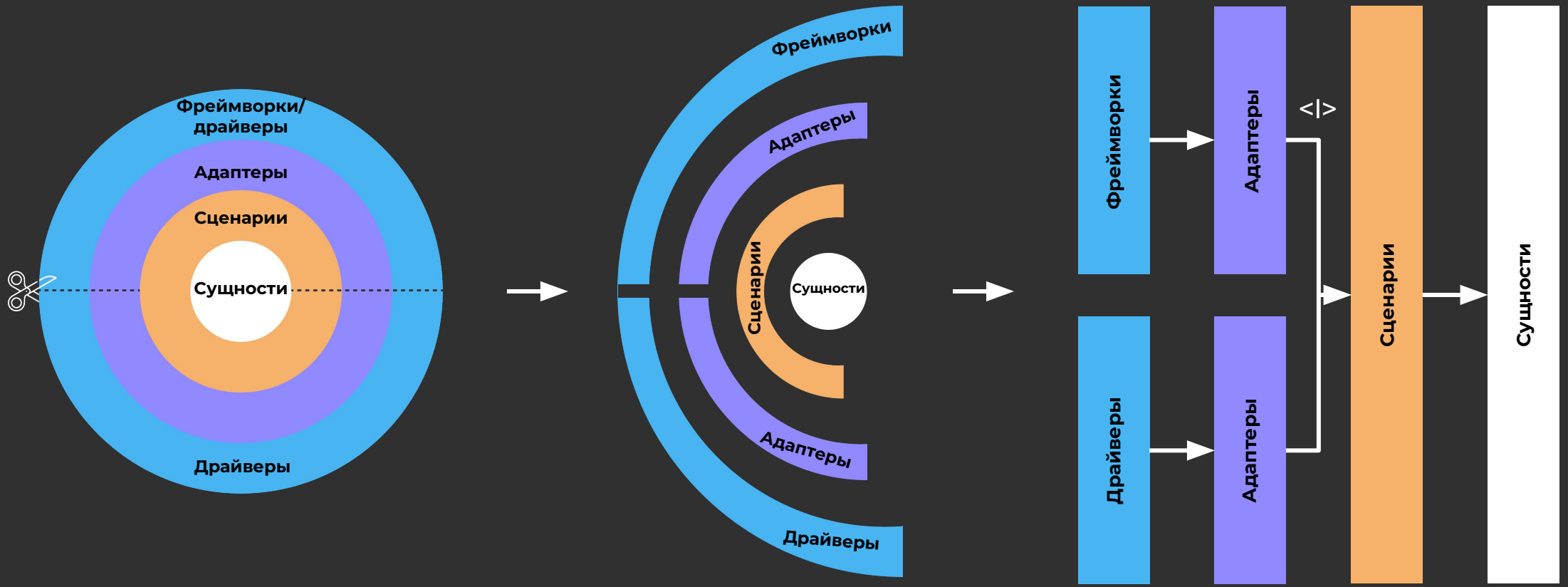
Слои чистой архитектуры

- Domain – Сущности
- Use Case – Сценарии
- Delivery / Repository – Адаптеры
- Drivers – Фреймворки и драйверы



Правило зависимостей

Зависимости в исходном коде должны быть направлены внутрь,
в сторону высокоуровневых политик



Сущности (Domain)

Сущности определяются бизнес-правилами предприятия:

- Реализует все случаи использования системы
  - Объект с методами
  - Структуры данных
  - Функции

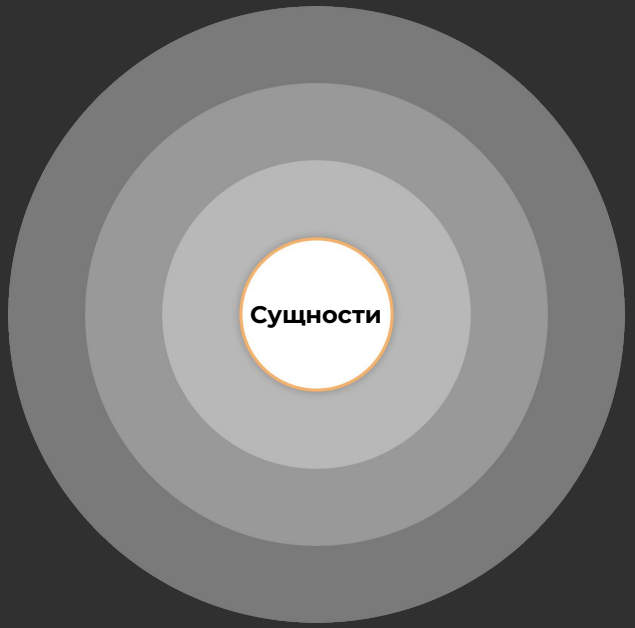
Не восприимчив к внешним изменениям:

- Базы данных
- Пользовательским интерфейсом
- Фреймворком

Изменения в работе сущности повлияет:

- Use Case – сценарии
- Delivery / Repository – адаптеры
- Drivers – Фреймворки / драйверы

Изменения поведения приложения затронут код в данном слое



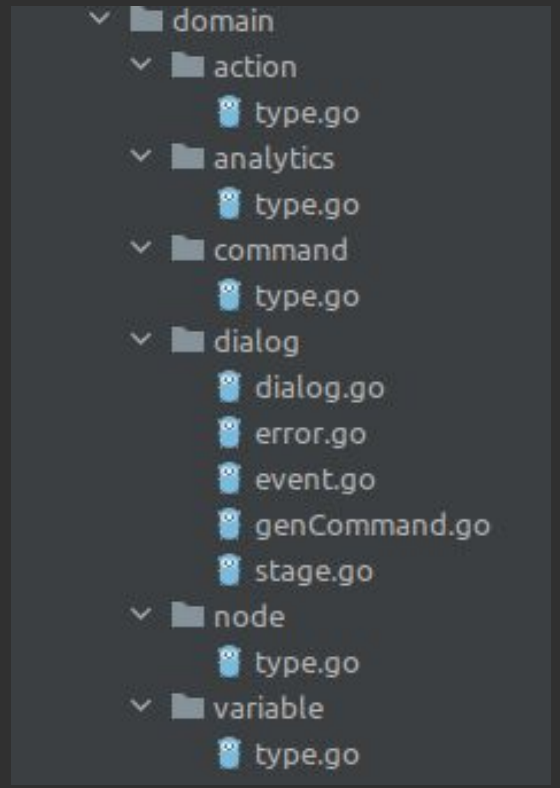
Сущности (Domain)

Примеры:

- Создание структур с которыми взаимодействует непосредственно пользователь
  - Раздел контактов – Контакт
  - Раздел группировки контактов – Группа
  - Раздел пользователя системы – Пользователь
- Проверка обязательности / формата полей
- Бизнес логика
  - При создании контакта, получать по номеру UTC
  - Заполнение ФИО из атомарных полей

Домен ничего не знает о других уровнях

Он содержит чистую бизнес-логику



Сценарии (Use Case)

Use case – это детализация, описание действия, которое может совершить пользователь системы

Использует domain, но ничего не знает о внешних слоях

Он понятия не имеет, вызывается ли:

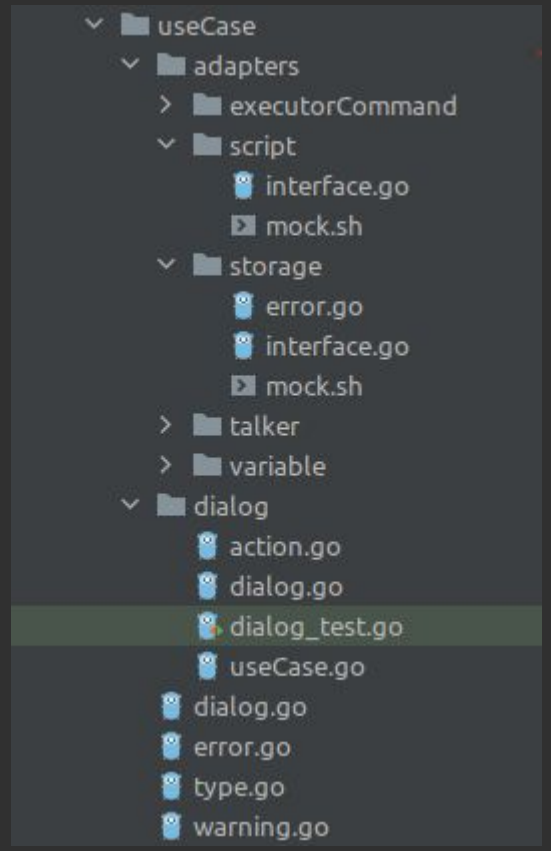
- HTTP-запросом
- Обработчиком Pub/Sub
- Командой CLI

Доступы к Repository / Delivery осуществляются при помощи interface который диктует Use case

Сценарии (Use Case)

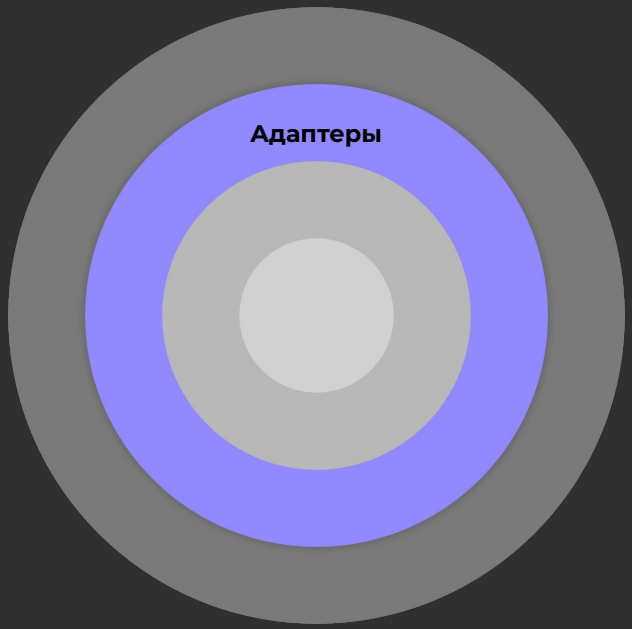
Примеры:

- Обработка команд полученных от Delivery
- Работа с другими Use Case
- Объединение данных с различных источников (Repository)
  - Базы данных
  - Сторонних сервисов
- Отправка команд / событий (Repository)
  - Сохранение данных
  - Генерация событий



Интерфейс-Адаптеры

Основная функция уровня интерфейсного адаптера – преобразование данных

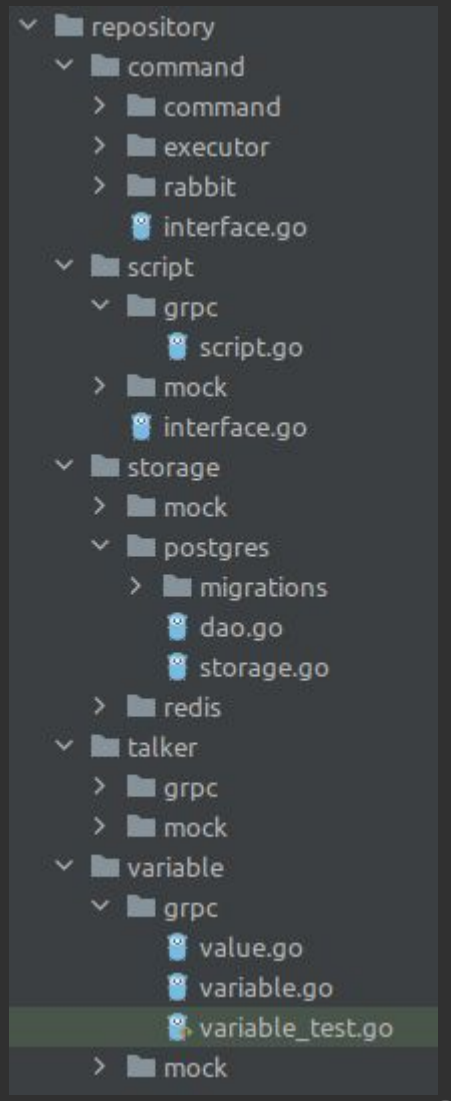


- Данные из внешних слоёв к удобному формату Use Case
- Данные из Use Case к удобному формату для внешнего слоя

Интерфейс-Адаптеры (Repository Pattern)

- Обеспечивает абстракцию данных через интерфейс
- Использование этого шаблона может помочь добиться слабой связи и сохранить объекты Domain

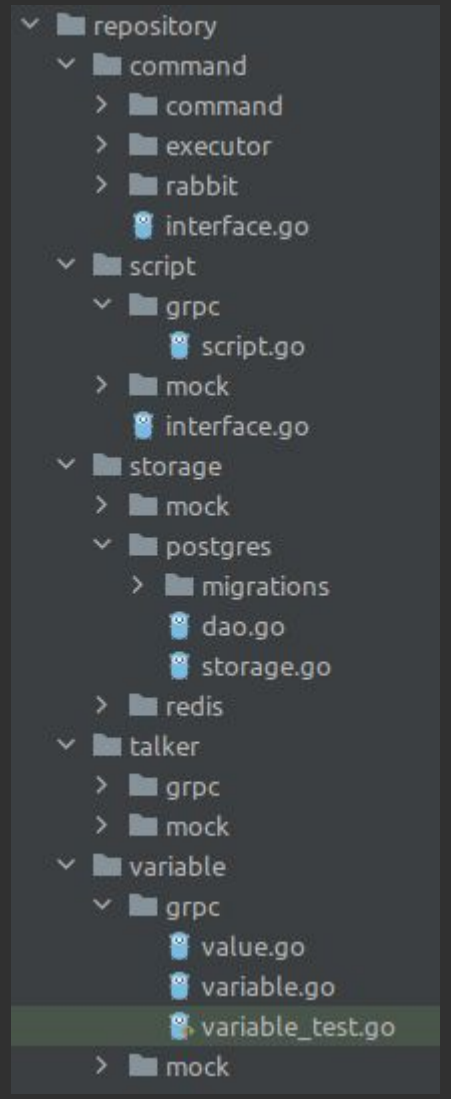
```go
type Storage interface {
    Contact
    Group
    Campaign
}

type Contact interface {
    CreateContact(ctx robo_context.Context, createdBy uuid.UUID, contacts ...*contact.Contact) ([]*contact.Contact, error)
    UpdateContact(ctx robo_context.Context, ID, createdBy uuid.UUID, updateFn func(c *contact.Contact) (*contact.Contact, error)) (*contact.Con
    DeleteContact(ctx robo_context.Context, createdBy uuid.UUID, filters filter.Filters) ([]*contact.Contact, error)

    ContactReader
}

type ContactReader interface {
    ListContact(ctx robo_context.Context, createdBy uuid.UUID, parameter queryParameter.QueryParameter) ([]*contact.Contact, error)
    ReadContactByID(ctx robo_context.Context, createdBy, ID uuid.UUID) (response *contact.Contact, err error)
    CountContact(ctx robo_context.Context, createdBy uuid.UUID, filters filter.Filters) (uint64, error)
    ReadContactByGroupID(ctx robo_context.Context, createdBy, groupID uuid.UUID, parameter queryParameter.QueryParameter) ([]*contact.Contact,
    ListUTCByAllContacts(ctx robo_context.Context, createdBy uuid.UUID) ([]*useCase.GroupContactsByUTC, error)

    FieldListContact(ctx robo_context.Context, createdBy uuid.UUID) ([]*contact.Field, error)
    GetContactFieldValue(ctx robo_context.Context, contactID uuid.UUID, field, fieldID string) (string, error)
}

type Group interface {
    CreateGroup(ctx robo_context.Context, createdBy uuid.UUID, group *group.Group) (*group.Group, error)
    UpdateGroup(ctx robo_context.Context, createdBy, ID uuid.UUID, updateFn func(group *group.Group) (*group.Group, error)) (*group.Group, erro
    ListGroup(ctx robo_context.Context, createdBy uuid.UUID, parameter queryParameter.QueryParameter) ([]*group.Group, error)
    ReadGroupByID(ctx robo_context.Context, createdBy, ID uuid.UUID) (*group.Group, error)
    DeleteGroup(ctx robo_context.Context, createdBy, ID uuid.UUID) error
    CountGroup(ctx robo_context.Context, createdBy uuid.UUID, filters filter.Filters) (uint64, error)
    ContactInGroup
}
```



Интерфейс-Адаптеры (Repository)

Примеры:

- Взаимодействие с БД
- Кэширование данных
- Отправка данных сторонним системам
  - REST
  - gRPC
- Преобразование данных полученных от сторонних к формату удобному для Use Case
  - Приведение к структурам Use Case
  - Приведение к структурам Domain
  - Приведение к структурам Сторонней системы

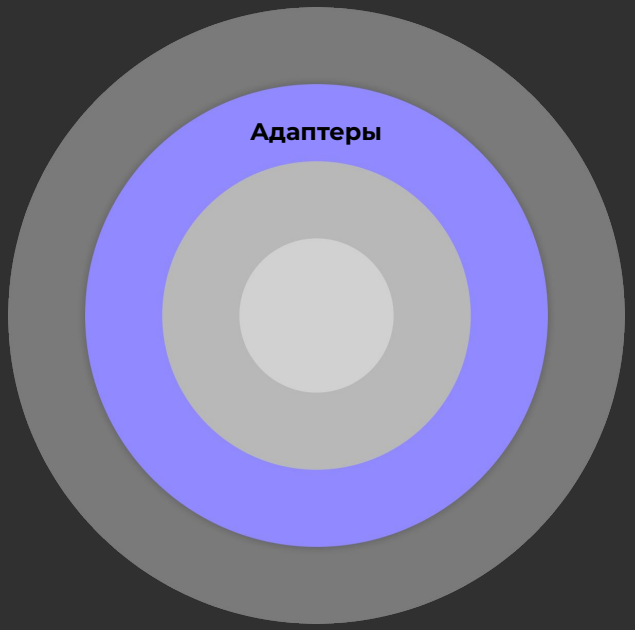


Интерфейс-Адаптеры (Delivery)

Единственная точка входа для сторонней системы

- Выставление внешнего API
- Работа с методами Use Case

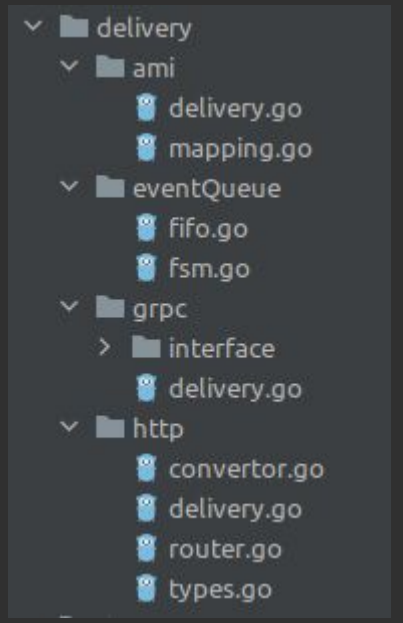
Не может напрямую обращаться к Repository

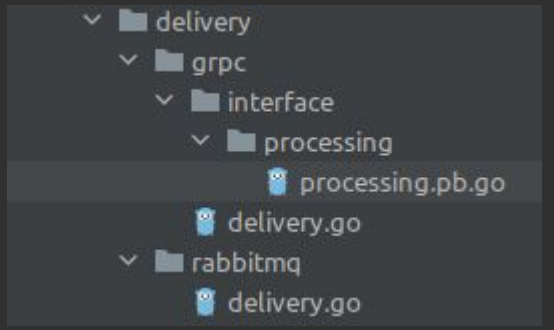


Интерфейс-Адаптеры (Delivery)

Примеры:

- Получение данных от сторонних систем
  - REST
  - gRPC
- Преобразование данных полученных от сторонних к формату удобному для Use Case
  - Приведение к структурам Use Case
  - Приведение к структурам Domain
- Формирование API для взаимодействия





Фреймворки и драйверы. (Drivers)

Самый внешний уровень модели

- Фреймворки
- Инструменты доступа к базам данных
- Веб-фреймворки

Пересечение границ

- Примитивные типы
- Простые структуры
- Data Transfer Objects (DTO)
- Вызовы функций через аргументы

Важно, чтобы через границы передавались простые, изолированные структуры данных

При передаче через границу данные всегда должны принимать форму, наиболее удобную для внутреннего круга.

Как это всё объединять?

```go
package main

import (
    "fmt"
    "net"
    "os"
    "os/signal"
    "syscall"

    "github.com/pkg/errors"
    "github.com/spf13/viper"

    "robovoice/micro-services/dialogProcessor/cmd/app/wire"
    grpcRabbitMQ "robovoice/micro-services/dialogProcessor/internal/processing/delivery/rabbitmq"
    "robovoice/micro-services/dialogProcessor/internal/processing/repository/command/executor"
    messageBroker "robovoice/micro-services/dialogProcessor/internal/processing/repository/command/rabbit"
    grpcScript "robovoice/micro-services/dialogProcessor/internal/processing/repository/script/grpc"
    storage "robovoice/micro-services/dialogProcessor/internal/processing/repository/storage/postgres"
    grpcTalker "robovoice/micro-services/dialogProcessor/internal/processing/repository/talker/grpc"
    grpcVariable "robovoice/micro-services/dialogProcessor/internal/processing/repository/variable/grpc"
    "robovoice/micro-services/dialogProcessor/internal/processing/useCase/dialog"
    "robovoice/pkg/di/robo_context"
    log "robovoice/pkg/di/robo_logger"
    "robovoice/pkg/messageBroker/rabbitMQ"
    "robovoice/proto"
    protoClumsyTalker "robovoice/protobuf/clumsyTalker"
    "robovoice/protobuf/contactServiceV2/contact"
    protoScriptAction "robovoice/protobuf/scriptService/action"
    protoMarker "robovoice/protobuf/scriptService/marker"
    protoScript "robovoice/protobuf/scriptService/script"
)
```

Пакет cmd
Функция main

Теория всё? Можно уже кодить?

Примеры кода на чистой архитектуре:

1. go-clean-template
2. go-clean-arch
3. go-clean-architecture

Блиц-опрос

1. *Что делать, если у вас Domain сущность имеет данные из двух разных систем?*

2. *Как быть, если вам нужно объединить данные из разных БД?*

3. *Как решить проблему транзакционности при вставке в несколько таблиц одно БД?*
   (Создание контакта и добавление его в группу)

Таков путь...

День 1:

1. Создание структуры проекта

День 2:

2. Подключение к БД
3. Чистая архитектура:
   - 3.1. Структура папок по чистой архитектуре
   - 3.2. Наполнение – Domain
   - 3.3. Интерфейс – Use Case
   - 3.4. Интерфейс – Repository
   - 3.5. Конструкторы слоёв
   - 3.6. Инициализация слоёв на main
   - 3.7. Реализация интерфейса – Use Case

День 3:

   - 3.8. Реализация интерфейса – Repository
   - 3.9. Реализация – Delivery
4. Использование – Context
5. Добавить логи
6. Добавить трассировку
7. Тестирование**

Вопросы