mtproject

maybetomorrow

Встреча 1. Результаты прошлого и новое будущее

Вам необходимо пройти небольшой опрос.
Мне важно понять реальную ситуацию, чтобы никого не напрягать лишними задачами и правильно распределить нагрузку. Прошу отвечать честно!

1. Отсутствие асинхронности
Один компонент ждет, пока будет готов другой.

2. Отсутствие четких технических требований
Указано название задачи, но нет конкретного пояснения.
Да, мы его обсуждаем на встречах, но через минуту уже забываем, а нигде записано этого не было.

3. Отсутствие документации
Чтобы понять, как работает та или иная фича, необходимо залазить в код.

4. Неудобство развертывания приложения
Если у кого-то нет постгреса, то ему приходится скачивать.
Если у кого-то разная версия го, то ему приходится скачивать.
Если мы добавим новый компонент, то его придется скачивать.

5. Отсутствие единой архитектуры
Один запрос возвращает строку, другой bool, третий JSON-объект

1. Новая работа

Не переписываем, а рефакторим итеративно: начинаем с новой структурой, старый код подтягиваем постепенно

2. Разобьем проект на две (?) части

Создан отдельный гитхаб аккаунт mtproject-ru (название нашей команды?), в котором будет два(?) репозитория: server - бэкенд, website - frontend - это повысит масштабируемость нашего проекта

> ! Важно, что maybetomorrow - это проект, а сама веб-платформа - один из продуктов данного проекта. Т.к. в будущем мы можем не остановиться на сайте, а сделать, например, мобильное приложение, то лучше с самого начала заложить правильную структуру. *(мое видение)*

3. Будем работать асинхронно

Я постараюсь сделать задачи так, чтобы все работы были не зависимы друг от друга.
Пример: на бэке нет функции получения пользователя, а на фронте уже будет готов компонент профиля с синтетическими данными

4. Тестирование

Да, скорее всего, придется писать тесты... (либо тыкать руками, пока не знаю).
Пример: бэк запушил фичу пользователя, на фронте компонент профиля готов, но при получении данных возникают ошибки. Наша задача не быстренько сделать фичу, а сделать качественно, пусть и с опозданием!

5. Feature-based архитектура (по фичам)

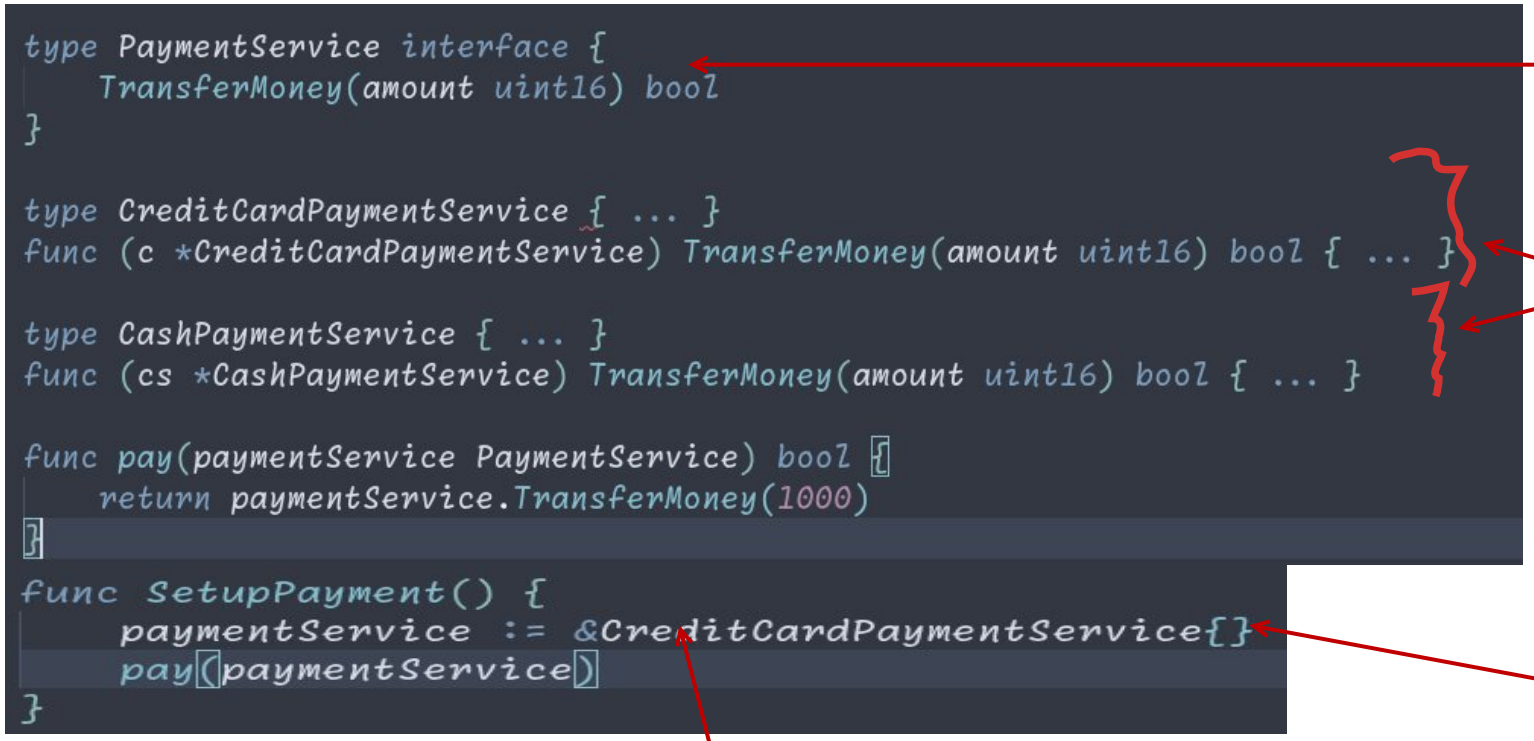
Если раньше мы разрабатывали каждый функционал отдельно, то сейчас весь функционал будет сгруппирован. Даже если на это будет уходить больше времени. Но это поможет в тестировании!
Пример: Фича events (события) - весь функционал БД по событиям, все эндпоинты, все страницы на фронте. Полное тестирование данной фичи, она полностью функционирует - забыли, идем дальше.

6. Онлайн встречи

Да, придется иногда проводить встречи онлайн, потому что оффлайн не всегда получается.
Будем использовать либо дискорд, либо Яндекс телемост

Фича - совокупность функционала, объединенного общим смыслом
(пр: фича пользователя - модель пользователя, репо, контроллера

В дальнейшем может встретиться большое количество новых слов и принципов, поэтому я их немного разбил. Здесь будут основы.



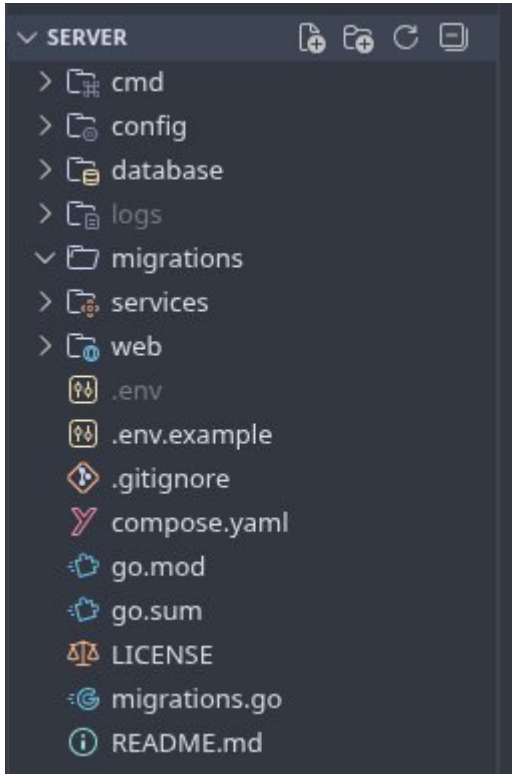
Интерфейс - интерфейс, описание функционала без реализации

Имплементация - реализация функционала, описанного в интерфейсе

Инстанс - объект имплементации

Имплементация, которая используется в данный момент, называется *стандартной*

В дальнейшем выражение «создаем инстанс» будет означать «создаем объект стандартной имплементации



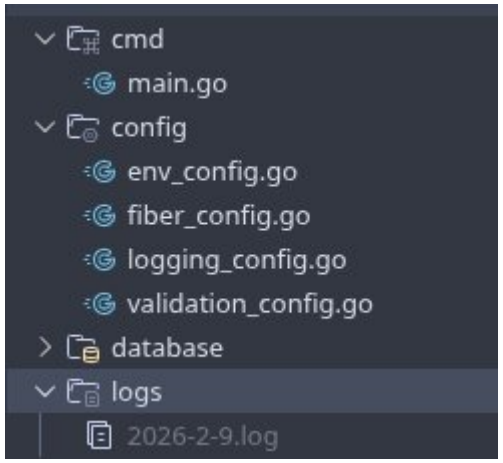
cmd - точка входа в приложение, там main.go
config - конфигурации
database - структуры БД, репозитории и модели
logs - логи приложения (.log или .txt файлы)
migrations - миграции для базы данных (используем goose)
services - сервисы
web - ендпоинты, контроллеры, DTO

.env - переменные окружения
.env.example - пример необходимых переменных
compose.yaml, Dockerfile - конфигурация докера*

Каждая папка, кроме cmd, config, logs, migrations представляет из себя «модуль». В каждом модуле есть файл, который имеет схожее с модулем название.
В нем находятся Setup-функции, которые собирают все предметы в модуле воедино

Модуль - совокупность схожего функционала (пр: модуль сервисов, модуль конфигурации)

Структура бэкенда



cmd/main.go

```go
func main() {
    config.SetupEnv()
    config.SetupLogging()

    database.SetupDatabase()
    server.SetupMigrations()

    app := config.SetupFiber()

    services.SetupServices(app)
    web.SetupWeb(app)

    port := os.Getenv("SERVER_PORT")
    if port == "" {
        slog.Info("SERVER_PORT не найдена в переменных окружения. Используем 3000 порт")
        port = ":3000"
    }

    log.Fatal(app.Listen(port))
}
```

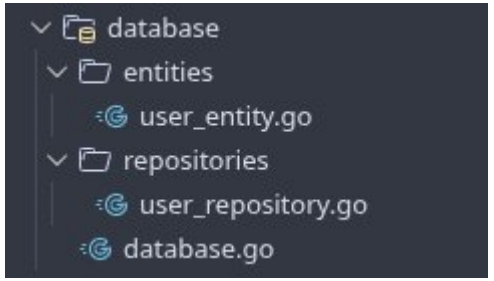
config/env_config.go

```go
func SetupEnv() {
    if err := godotenv.Load(); err != nil {
        log.Fatalf("Не удалось загрузить .env файл: %v", err)
    }
}
```

logs/year_month_day.log

```
time=2026-02-09T20:29:12.996+07:00 level=INFO msg="Успешно подключились к базе данных"
time=2026-02-09T20:29:13.009+07:00 level=INFO msg="Успешно создали все необходимое в базе данных"
time=2026-02-09T20:32:36.638+07:00 level=INFO msg="Успешно подключились к базе данных"
time=2026-02-09T20:32:36.641+07:00 level=INFO msg="Успешно создали все необходимое в базе данных"
time=2026-02-09T20:32:42.562+07:00 level=ERROR msg="Пользователь с именем niikage не найден"
time=2026-02-09T20:33:40.959+07:00 level=INFO msg="Успешно подключились к базе данных"
time=2026-02-09T20:33:40.963+07:00 level=INFO msg="Успешно создали все необходимое в базе данных"
time=2026-02-09T20:33:45.223+07:00 level=ERROR msg="nil pointer passed to StructScan destination"
```

Логи в гит не пушим! Они только для локальной разработки и сервера, не для репозитория!!!



Почему здесь есть log и slog?

У нас будет договоренность (далее), что из модуля **log** вызываем только **Fatalf** и завершаем приложение
Используем только при **критических** ошибках.

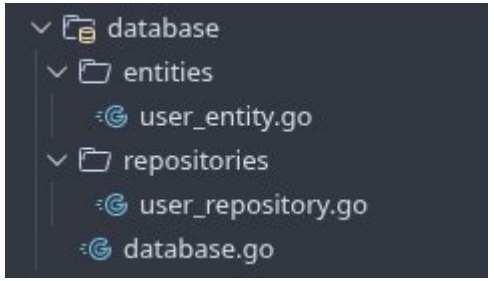
Если же возникает некритичная ошибка или предупреждение, то мы используем **slog**, который выводи ее в консоль и в ЛОГ ФАЙЛ

[Ссылка на slog]

```go
import (
    "log"
    "log/slog"
    "os"

    _ "github.com/jackc/pgx/v5/stdlib"
    "github.com/jmoiron/sqlx"
)

var DB *sqlx.DB

func SetupDatabase() {
    connectionString := os.Getenv("DATABASE_CONNECTION_STRING")
    if connectionString == "" {
        log.Fatal("DATABASE_CONNECTION_STRING не найдено в переменных окружения")
    }

    var err error
    DB, err = sqlx.Connect("pgx", connectionString)
    if err != nil {
        log.Fatalf("Не удалось подключиться к базе данных: %v", err)
    }

    slog.Info("Успешно подключились к базе данных")
    if err := initSchema(DB); err != nil {
        log.Fatal(err)
    }
    slog.Info("Успешно создали все необходимое в базе данных")
}
```

/database.go

Структура бэкенда. Модуль БД



/entities/user_entity.go

Модель - объект базы данных

```go
package entities

type UserEntity struct {
    Id       uint   `db:"id"`
    Username string `db:"username"`
    Password string `db:"password"`
}
```

```go
type UserRepository interface {
    GetUserByUsername(username string) (*entities.UserEntity, error)
}

type UserRepositoryImpl struct{}

func (u *UserRepositoryImpl) GetUserByUsername(username string) (*entities.UserEntity, error) {
    var user entities.UserEntity

    err := database.DB.Get(&user, "SELECT * FROM users WHERE username=$1", username)
    if err != nil {
        return nil, err
    }

    return &user, nil
}
```

/repositories/user_repository.go

Репозиторий(репо) - интерфейс и его стандартная имплементация для работы с БД.
Он принимает только модели (или их поля) и возвращает только их же

Весьма интересный подход, не так ли? Зачем нам интерфейс, потом какая то структура, а потом реализация методов интерфеса для этой структуры, если можно просто сделать структуру, без интерфейса?

```go
type UserRepository interface {
    GetUserByUsername(username string) (*entities.UserEntity, error)
}

type UserRepositoryImpl struct{}

func (u *UserRepositoryImpl) GetUserByUsername(username string) (*entities.UserEntity, error) {
    var user entities.UserEntity

    err := database.DB.Get(&user, "SELECT * FROM users WHERE username=$1", username)
    if err != nil {
        return nil, err
    }

    return &user, nil
}
```

/repositories/user_repository.go

Этот подход называется *Dependency Inversion* или *Внедрение зависимости через интерфейс* и является частью принципов SOLID и DI (dependency injection, внедрение зависимостей).

Причина	Зачем
Тестирование	Можно подменять реализации
DI-контейнеры	Интерфейс – естественный ключ для внедрения зависимостей
Слабая связанность	Зависим от абстракции, а не от конкретной реализации

Пример:

```
type PaymentService interface {
    TransferMoney(amount uint16) bool
}

type CreditCardPaymentService { ... }
func (c *CreditCardPaymentService) TransferMoney(amount uint16) bool { ... }

type CashPaymentService { ... }
func (cs *CashPaymentService) TransferMoney(amount uint16) bool { ... }

func pay(paymentService PaymentService) bool {
    return paymentService.TransferMoney(1000)
}
```

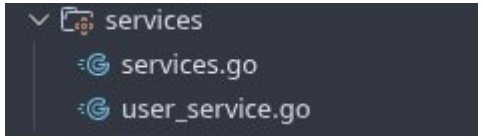
Здесь нам не важна внутренняя
реализация функции, самое главное, что она есть!
Поэтому в функцию pay мы можем передать или CreditCardPaymentServic, или CashPaymentService - нам без разницы!

Наглядное применение будет далее.

год месяц день время_название миграции.sql

```sql
-- +goose Up
-- +goose StatementBegin
CREATE TABLE IF NOT EXISTS users (
    id INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    username VARCHAR(32) UNIQUE NOT NULL,
    password VARCHAR(32) NOT NULL
);

CREATE INDEX IF NOT EXISTS user_username_idx ON users(username);
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
DROP TABLE TABLE IF EXISTS users;
-- +goose StatementEnd
```

Миграция - изменение базы данных (добавление строк, изменение колонок и т.п.)



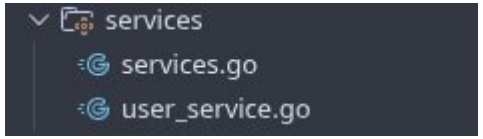
/services.go

```go
func SetupServices(app *fiber.App) {
    userRepository := &repositories.UserRepositoryImpl{}
    userService := &UserServiceImpl{
        repo: userRepository,
    }

    app.State().Set(USER_SERVICE_KEY, userService)
}
```

Тут мы создаем инстанс репозитория и сервиса, после чего передаем уже созданный репозиторий в реализацию, добавляем в состояние (хранилище) приложения.

Получится такая ситуация, что и репозиторий, и сервис будут созданы ТОЛЬКО ОДИН РАЗ и будут переиспользоваться без потерь, что очень даже хорошо. *(пример позже)*

Структура бэкенда. Сервисы




```go
package services

import (
    "fmt"

    "github.com/mtproject-ru/server/database/entities"
    "github.com/mtproject-ru/server/database/repositories"
)

const USER_SERVICE_KEY = "user-service"

type UserService interface {
    GetUserByUsername(username string) (*entities.UserEntity, error)
}

type UserServiceImpl struct {
    repo repositories.UserRepository
}

func (u *UserServiceImpl) GetUserByUsername(username string) (*entities.UserEntity, error) {
    user, err := u.repo.GetUserByUsername(username)
    if err != nil {
        return nil, fmt.Errorf("Пользователь с именем %s не найден", username)
    }
    return user, nil
}
```

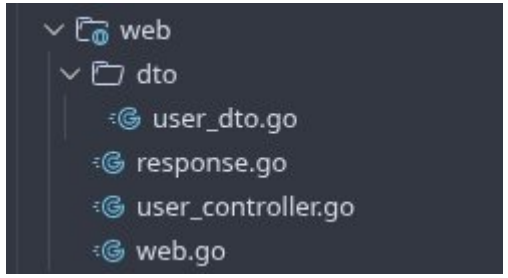
/user_service.go

Каждый сервис имеет СЕРВИС_SERVICE_KEY с уникальным названием сервиса (для DI)

Ну и та же чехарда с интерфейсом и имплементацией

Сервис - интерфейс и его стандартная имплементация для связывания контроллера с репозиторием.
Он занимается обработкой входящих данных, передает их в репозиторий и возвращает результат

Связь с репозиторием осуществляется через поле repo в имплементации

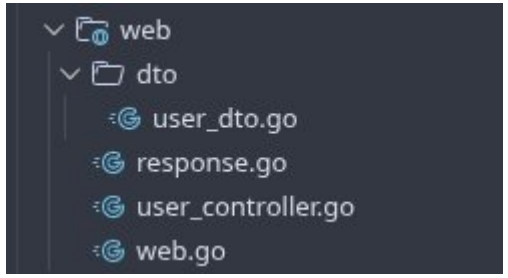


/dto/user_dto.go

```go
package dto

import "github.com/mtproject-ru/server/database/entities"

type UserResponse struct {
    Id       uint   `json:"id"`
    Username string `json:"username"`
}

// Mappers
func ToUserResponse(entity *entities.UserEntity) UserResponse {
    return UserResponse{
        Id:       entity.Id,
        Username: entity.Username,
    }
}
```

ДТО(дэтэошка) - объект для передачи данных в контроллеры (и некоторые функции)

Пример: мы получаем пользователя из бд и хотим его вернуть в ответе, но мы же не можем возвращать пароль пользователя на запрос простого описания. DTO, по своей сути, копия модели, только без «ненужных» полей, но ее уникальность в том, что она редко изменяется даже при частом изменений модели.



/response.go

Что такое Response?

В проблемах было указано отсутствие единой архитектуры, а это ее решение - единый формат ответа. Не зависимо от того, что ендпоинт возвращает, он всегда возвращает Response в теле ответа.
Нужно строку = Data: string
Нужно объект = Data: fiber.Map
Нужно ничего = Data: nil (null) - *не рекомендовано*
И т.п.

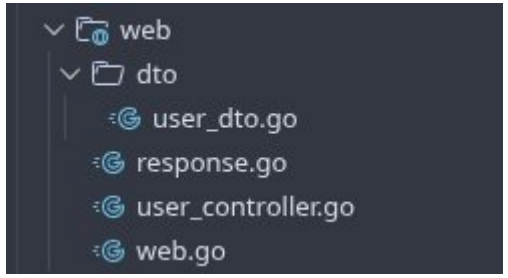
```go
package web

type Response struct {
    Ok    bool  `json:"ok"`
    Data  any   `json:"data"`
    Error error `json:"error"`
}

func NewResponse(data any, err error) Response {
    if err != nil {
        return Response{
            Ok:    false,
            Data:  nil,
            Error: err,
        }
    }

    return Response{
        Ok:    true,
        Data:  &data,
        Error: nil,
    }
}
 func NewOkResponse(data any) Response {
     return Response{
         Ok:    true,
         Data:  data,
         Error: nil,
     }
 }

 func NewErrResponse(err error) Response {
     return Response{
         Ok:    false,
         Data:  nil,
         Error: err,
     }
 }
```

/api/users/create -> handleCreateUser(...) error {...}

Эндпоинт - конечный адрес, на который отправляется запрос
Группа - совокупность эндпоинтов
Хэндлер - функция, которая обрабатывает запрос на определенный эндпоинт

Контроллер - совокупность хэндлеров, объединенных одной группой (пр: контроллер пользователей для всех /api/users/**).
Принимает запрос, передает его в сервис, возвращает ответ

Структура бэкенда. Веб



/web.go

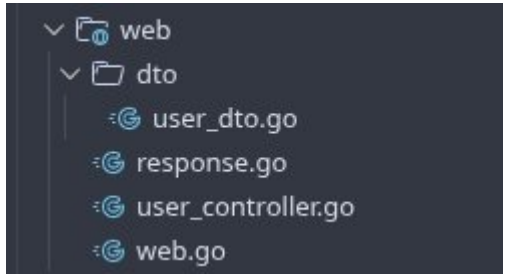
```go
package web

import (
    "github.com/gofiber/fiber/v3"
)

func SetupWeb(app *fiber.App) {
    users := app.Group("/api/users")
    setupUserController(users)
}
```

Про setupUserController:
Это Setup-функция, принимает в себя fiber.Router - роутер именно для группы.
В web.go видно, что мы сначала создаем группу users (/api/users), а потом ее передаем в setupUserController. Таким образом получаем вот это.

```go
func setupUserController(router fiber.Router) {
    router.Get("/:username", getUserByUsernameHandler)
}

func getUserByUsernameHandler(ctx fiber.Ctx) error {
    username := ctx.Params("username")
    if username == "" {
        err := errors.New("Не указано имя пользователя для поиска")
        return ctx.Status(fiber.StatusBadRequest).JSON(NewErrResponse(err))
    }

    userService := fiber.MustGetState[services.UserService](
        ctx.App().State(),
        services.USER_SERVICE_KEY
    )
    user, err := userService.GetUserByUsername(username)

    if err != nil {
        slog.Error(err.Error())
        return ctx.Status(fiber.StatusNotFound).JSON(NewErrResponse(err))
    }

    return ctx.Status(fiber.StatusOK).JSON(NewOkResponse(
        dto.ToUserResponse(user),
    ))
}
```

Вот тут и играет наш DI и танцы с интерфейсами.
Мы получаем сервис пользователя, но не структуру (UserServiceImpl), а ИНТЕРФЕЙС.
Здесь нам не важна имплементация UserServiceImpl1 или UserServiceImpl2 - в этом и прелесть.
Пример далее

/user_controller.go



Пример ближе к нам

Не обращаем внимание на квадратные скобки у интерфейса - это дженерики, мы не о них говорим

```go
type JustProfile struct { /* ... */ }
type ProfileWithDevices struct { /* ... */ }

type UserService[T JustProfile | ProfileWithDevices] interface {
    GetUserByUsername(username string) T
}

type UserServiceImpl struct { /* ... */ }
func (u *UserServiceImpl) GetUserByUsername(username string) JustProfile { /* ... */ }

type AdminServiceImpl struct { /* ... */ }
func (a *AdminServiceImpl) GetUserByUsername(username string) ProfileWithDevices { /* ... */ }
```

1. Разделение властей
Если в модуле, который писал другой человек, необходимо внести изменения, то лучше попросить об этом самого разработчика модуля

2. Используем пропогирование ошибок
Если ошибка возникла в репозитории, то выбрасываем ее в сервис. В сервисе обрабатываем эту ошибку, если она влияет на результат работы сервиса, то создаем и пробрасываем в контроллер новую ошибку. В контроллере, если ошибка достаточно важная и/или редкая, то логируем (slog.Warn или slog.Error) и возвращаем ошибочный респонс СО СТАТУСОМ!

3. Правильные ошибки
Все ошибки создаем «на месте поимки» с помощью errors.New или fmt.Errorf.
Если ошибка встречается много раз (пр: NotFoundError для пользователя, события и т.п.), то выносим в отдельный модуль и создаем кастомную ошибку

4. Логическая группировка кода
Не писать всю логику для событий, пользователей, создания календаря и обработки строк к одном файле - лучше разбить на несколько. Для фичей используется вышеупомянутая структура.
Если утилита может использовтся во многих частях приложения, то выносим в отдельный модуль. Иначе пишем в файле, где эта утилита нужна, делая функцию приватной

5. Инкапсулируем код
Если ваша функция нужна только в данном файле или данном пакете (pakage), то делаем ее приватной, чтобы другие модули не имели к ней доступ (а зачем она им, вдруг напакостят?)

6. Нормальные имена
Если из названия функции или переменной не понятно, что она делает или за что отвечает, то пишем небольшую документацию. Не боимся давать длинные имена, но если прям совсем длинно получается, то выделяем основную суть, а остальное пишем в документации.
Пример: func GetTime(s time.Time, e time.Time) time.Time - не особо понятно, что за время мы должны передать и какое получим, поэтому, либо переименовываем (func GetEventDuration(start ..., end ...) ..., либо небольшую документацию *(но в этом примере прям вообще кринж, лучше нормально называть)*

7. Документация для будущих поколений (или нынешних)
Если функция сложна для понимания, то пишем небольшие пояснения к неясным блокам.
Автору кода может быть все понятно, но если я приду, прочитаю функцию и не пойму, что она делает и как работает, то капец.
Да и автор через неделю может забыть уже алгоритм, по которому писал функцию

8. Логирование для анализа ошибок

Если возникает серьезная ситуация(пр1) или редкая ошибка(пр2), то используем slog.{Info, Warn, Error}.
А так же, если происходит какое то важное и опасное действие, то тоже логируем, даже если нет ошибки (пр3)
В эти функции можно передавать дополнительные параметры - можно не бояться передать туда имя пользо или его айди.

Пример1: ... Пользователь с id = 123 пытался зайти в админ-панель без прав администратора. Неверный пароль...
Вот с такими данными будет легде понять что пытался сделать пользователь и кого надо банить. Тут мы используем slog.Warn

Пример2: ... Пользователь с id = 1234 зашел в админ-панель без прав администратора. Но мы его кикнули
Что? Ну вот тут slog.Error

Пример3: ... Администратор с id = 12345 зашел в админ-панель...
Тут ошибки нет, но лучше логировать slog.Info, так как достаточно важное действие

9. Используем двойную валидацию

Когда с фронта поступает какой либо объект в запросе (структура), то он, *скорее всего*, будет отвалидирован (проверен на соответствие ограничениям и стандартам), но лучше провалидировать и на бэкенде во второй раз. Для этого можно использовать пакет validator (ссылка) - он настроен в config/validation_config.go

Почему? Безопасность: запрос может прийти не только с фронта, но и из curl, скрипта, атаки.

Пример валидации:

```
10    // User contains user information
11    type User struct {
12        FirstName      string     `validate:"required"`
13        LastName       string     `validate:"required"`
14        Age            uint8      `validate:"gte=0,lte=130"`
15        Email          string     `validate:"required,email"`
16        Gender         string     `validate:"oneof=male female prefer_not_to"`
17        FavouriteColor string     `validate:"iscolor"`                // alias for 'hexcolor|rgb|rgba|hsl|hsla'
18        Addresses      []*Address `validate:"required,dive,required"` // a person can have a home and cottage...
19    }
```

В дальнейших презентациях и иных коммуникациях будут использовтся следующие термины:

Интерфейс - интерфейс, описание функционала без реализации
Имплементация - реализация функционала, описанного в интерфейсе
Инстанс - объект имплементации
Модель - объект базы данных
Миграция - изменение базы данных (добавление строк, изменение колонок и т.п.)
ДТО(дэтэошка) - объект для передачи данных в контроллеры (и некоторые функции)
Эндпоинт - конечный адрес, на который отправляется запрос (пр: /api/users/create)
Группа - совокупность эндпоинтов
Хэндлер - функция, которая обрабатывает запрос на определенный эндпоинт
Контроллер - совокупность хэндлеров, объединенных одной группой (пр: контроллер пользователей для всех /api/users/**). Принимает запрос, передает его в сервис, возвращает ответ
Репозиторий(репо) - интерфейс и его стандартная имплементация для работы с БД.
Он принимает только модели и возвращает только их же
Сервис - интерфейс и его стандартная имплементация для связывания контроллера с репозиторием.
Он занимается обработкой входящих данных, передает их в репозиторий и возвращает результат
Модуль - совокупность схожего функционала (пр: модуль сервисов, модуль конфигурации)
Фича - совокупность функционала, объединенного общим смыслом
(пр: фича пользователя - модель пользователя, репо, контроллера

goose - Goose is a database migration tool. Both a CLI and a library.
https://github.com/pressly/goose

log/slog - Package slog provides structured logging, in which log records include a message, a severity level, and various other attributes expressed as key-value pairs.
https://pkg.go.dev/log/slog

pgx - pgx is a pure Go driver and toolkit for PostgreSQL.
https://github.com/jackc/pgx

sqlx - sqlx is a library which provides a set of extensions on go's standard database/sql library.
https://github.com/jmoiron/sqlx

docker compose - With Docker Compose you use a YAML configuration file, known as the Compose file, to configure your application’s services, and then you create and start all the services from your configuration with the Compose CLI.
https://habr.com/ru/companies/ruvds/articles/450312/

docker (Dockerfile) - A Dockerfile is a key component in containerization, enabling developers and DevOps engineers to package applications with all their dependencies into a portable, lightweight container.
https://dev.to/prodevopsguytech/writing-a-dockerfile-beginners-to-advanced-31ie