|  |  |
| --- | --- |
| ДИСЦИПЛИНА | Технологии индустриального программирования |
| ИНСТИТУТ | Институт перспективных технологий и индустриального программирования |
| КАФЕДРА | Кафедра индустриального программирования |
| ВИД УЧЕБНОГО МАТЕРИАЛА | Практическое занятие |
| ПРЕПОДАВАТЕЛЬ | Адышкин Сергей Сергеевич |
| СЕМЕСТР | 2 семестр, 2024-2025 гг. |

**Практическое занятие №2**

**Структура Go-проекта**

**Цели:**

* Понять назначение ключевых директорий (cmd/, internal/, pkg/ и др.).
* Научиться раскладывать код и артефакты проекта по «правильным» местам.
* Собрать минимальный скелет проекта и запустить «пустой» main.go.

**Теоретическая часть**

В качестве примера будем использовать *project-layout*.

**Важно**: project-layout − **не официальный стандарт**, а набор распространённых практик сообщества. Его надо адаптировать под размер и цели проекта. Для учебных и маленьких проектов часто достаточно одного main.go и go.mod. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout))

**Коротко о главных папках**

| **Папка** | **Для чего нужна** | **Когда использовать** |
| --- | --- | --- |
| cmd/<app> | Входные точки приложений (каждая подпапка — отдельный бинарь). Внутри — минимальный main, который вызывает код из internal/ или pkg/. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Когда у проекта один или несколько исполняемых файлов (CLI, сервер). |
| internal/ | Приватный код (компилятор Go запрещает импортировать его извне). Можно делить на internal/app/<app> и internal/pkg/ для общего приватного кода. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Когда есть логика, которую не должны импортировать сторонние проекты. |
| pkg/ | Публичные библиотеки, которые **можно** импортировать из других проектов. Паттерн распространён, но не всеми рекомендован; для маленьких проектов не обязателен. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout), [appliedgo.com](https://appliedgo.com/blog/go-project-layout?utm_source=chatgpt.com)) | Когда у проекта много не-Go артефактов в корне и/или вы целенаправленно публикуете библиотеки. |
| vendor/ | Зависимости, «вендоренные» в репозиторий (go mod vendor). Нужен редко (есть модульный прокси). ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Если корпоративные требования запрещают внешние загрузки в CI/проде. |
| api/ | Контракты: OpenAPI/Swagger, protobuf, JSON-схемы. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Когда есть внешний API/IDL. |
| web/ | Фронтенд/шаблоны/статические файлы для веб-части. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Когда бэкенд обслуживает веб-ресурсы. |
| configs/ | Шаблоны конфигов по умолчанию. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Когда нужны примеры/дефолты конфигураций. |
| scripts/ | Скрипты сборки, анализа, локальных задач (держать Makefile «тонким»). ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Когда появляется рутинь на bash/PowerShell. |
| build/ | Описание упаковки (Dockerfile, deb/rpm), CI-скрипты. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Для деплоя и CI/CD. |
| deployments/ | Шаблоны развёртывания: docker-compose, Kubernetes/Helm, Terraform. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Когда готовите окружения. |
| test/ | Внешние тест-утилиты/данные (можно testdata/). ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Для интеграционных тестов, фикстур. |
| docs/, examples/, tools/, third\_party/, assets/, githooks/, website/ | Документация, примеры, вспомогательные тулзы, внешние утилиты, медиа-активы, гит-хуки, сайт проекта. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | По необходимости. |
| src/ | **Не использовать** в корне проекта (это «джавовская» привычка и путает с $GOPATH/src). ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout)) | Не нужно. |

*Небольшая полемика в сообществе:* часть гоферов предпочитает держать всё в internal/ и cmd/ без pkg/; другие — используют pkg/ как «витрину» публичных пакетов. Оба подхода жизнеспособны — выбираем осознанно. ([alexedwards.net](https://www.alexedwards.net/blog/11-tips-for-structuring-your-go-projects?utm_source=chatgpt.com), [appliedgo.com](https://appliedgo.com/blog/go-project-layout?utm_source=chatgpt.com))

**Рекомендуемая «минималка» для учебного проекта**

Для простых учебных проектов достаточно:

yourapp/

├─ cmd/

│ └─ yourapp/

│ └─ main.go

├─ internal/

│ └─ app/

│ └─ server/

│ └─ server.go (позже добавим)

├─ go.mod

└─ README.md

* main.go — только парсит конфиг/порт и вызывает internal/app/server.Start().
* Внутри internal/… позже появятся хендлеры, сервисы, репозитории — когда доберёмся до HTTP/БД.
* pkg/ пока **не создаём** (чтобы не усложнять). Такой «плоский» старт советуют и практики. ([alexedwards.net](https://www.alexedwards.net/blog/11-tips-for-structuring-your-go-projects?utm_source=chatgpt.com))

**Практическая часть**

**Что должно быть установлено**

* Go ≥ 1.21 (проверка: go version)
* Git (проверка: git --version)
* Любая IDE (VS Code / GoLand). Для VS Code: расширение “Go” от Google.

**1) Создаём скелет проекта**

**1.1. Инициализация**

**Windows (PowerShell):**

mkdir myapp; cd myapp

git init

go mod init github.com/<user>/myapp

**macOS/Linux (bash/zsh):**

mkdir -p myapp && cd $\_

git init

go mod init github.com/<user>/myapp

Добавьте .gitignore:

/bin/

/dist/

/coverage.out

\*.log

**1.2. Директории и файлы**

mkdir -p cmd/myapp internal/app utils

Создаём три файла:

cmd/myapp/main.go — минимальный вход.

internal/app/app.go — «сердце» приложения.

utils/logger.go — простой логгер.

Содержимое:

cmd/myapp/main.go

package main

import "github.com/<user>/myapp/internal/app"

func main() {

app.Run()

}

internal/app/app.go

package app

import (

"encoding/json"

"fmt"

"net/http"

"time"

"github.com/<user>/myapp/utils"

)

type pingResp struct {

Status string `json:"status"`

Time string `json:"time"`

}

func Run() {

mux := http.NewServeMux()

// Корневой маршрут

mux.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {

utils.LogRequest(r)

w.Header().Set("Content-Type", "text/plain; charset=utf-8")

fmt.Fprintln(w, "Hello, Go project structure!")

})

// Пример JSON-ручки: /ping

mux.HandleFunc("/ping", func(w http.ResponseWriter, r \*http.Request) {

utils.LogRequest(r)

w.Header().Set("Content-Type", "application/json; charset=utf-8")

\_ = json.NewEncoder(w).Encode(pingResp{

Status: "ok",

Time: time.Now().UTC().Format(time.RFC3339),

})

})

utils.LogInfo("Server is starting on :8080")

if err := http.ListenAndServe(":8080", mux); err != nil {

utils.LogError("server error: " + err.Error())

}

}

utils/logger.go

package utils

import (

"fmt"

"net/http"

"time"

)

func LogRequest(r \*http.Request) {

fmt.Printf("[%s] %s %s %s\n",

time.Now().Format(time.RFC3339),

r.RemoteAddr,

r.Method,

r.URL.Path,

)

}

func LogInfo(msg string) {

fmt.Printf("[INFO] %s %s\n", time.Now().Format(time.RFC3339), msg)

}

func LogError(msg string) {

fmt.Printf("[ERROR] %s %s\n", time.Now().Format(time.RFC3339), msg)

}

На этом этапе у вас уже есть минимальный сервис с двумя ручками: / (текст) и /ping (JSON).

**2) Запуск и быстрая проверка**

**2.1. Запуск (без сборки)**

go run ./cmd/myapp

Откройте в браузере:

http://localhost:8080/ → текст “Hello…”

http://localhost:8080/ping → {"status":"ok","time":"..."}

В терминале видны логи запросов.

**2.2. Проверка через curl**

curl -i http://localhost:8080/

curl -i http://localhost:8080/ping

**2.3. Сборка бинарника (опционально)**

**Windows:**

go build -o bin\myapp.exe ./cmd/myapp

.\bin\myapp.exe

**macOS/Linux:**

go build -o bin/myapp ./cmd/myapp

./bin/myapp

**3) Мини-упражнения (закрепление)**

**Упр. А — Request-ID и улучшение логов**

Добавьте простой request-id (генерируется, если отсутствует заголовок X-Request-Id) и прокидывайте его в ответ.

В utils/logger.go добавим генератор:

package utils

import (

"crypto/rand"

"encoding/hex"

// ...

)

func NewID16() string {

b := make([]byte, 8)

\_, \_ = rand.Read(b)

return hex.EncodeToString(b)

}

В internal/app/app.go — middleware-обёртка:

func withRequestID(next http.Handler) http.Handler {

return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {

id := r.Header.Get("X-Request-Id")

if id == "" {

id = utils.NewID16()

}

w.Header().Set("X-Request-Id", id)

next.ServeHTTP(w, r)

})

}

И применим её при запуске сервера:

mux := http.NewServeMux()

// ... handle funcs как раньше

handler := withRequestID(mux)

utils.LogInfo("Server is starting on :8080")

if err := http.ListenAndServe(":8080", handler); err != nil {

utils.LogError("server error: " + err.Error())

}

Проверьте:

curl -i http://localhost:8080/ping

curl -i -H "X-Request-Id: demo-123" http://localhost:8080/ping

В ответе появится заголовок X-Request-Id.

**Упр. B — Единый формат JSON-ошибок**

Создайте утилиту для ответов в JSON:

utils/httpjson.go

package utils

import (

"encoding/json"

"net/http"

)

type JSONError struct {

Error string `json:"error"`

}

func WriteJSON(w http.ResponseWriter, code int, v any) {

w.Header().Set("Content-Type", "application/json; charset=utf-8")

w.WriteHeader(code)

\_ = json.NewEncoder(w).Encode(v)

}

func WriteErr(w http.ResponseWriter, code int, msg string) {

WriteJSON(w, code, JSONError{Error: msg})

}

И добавьте новый маршрут с ошибкой (в app.go, рядом с /ping):

mux.HandleFunc("/fail", func(w http.ResponseWriter, r \*http.Request) {

utils.LogRequest(r)

utils.WriteErr(w, http.StatusBadRequest, "bad\_request\_example")

})

Проверьте:

curl -i http://localhost:8080/fail

**Упр. C — Разделение обработчиков**

Создайте пакет для хендлеров:

mkdir -p internal/app/handlers

internal/app/handlers/ping.go

package handlers

import (

"encoding/json"

"net/http"

"time"

"github.com/<user>/myapp/utils"

)

type pingResp struct {

Status string `json:"status"`

Time string `json:"time"`

}

func Ping(w http.ResponseWriter, r \*http.Request) {

utils.LogRequest(r)

w.Header().Set("Content-Type", "application/json; charset=utf-8")

\_ = json.NewEncoder(w).Encode(pingResp{

Status: "ok",

Time: time.Now().UTC().Format(time.RFC3339),

})

}

И подправьте регистрацию маршрутов в app.go:

import "github.com/<user>/myapp/internal/app/handlers"

// ...

mux.HandleFunc("/ping", handlers.Ping)

**4) VS Code — удобные мелочи (опционально)**

1. Установите расширение **Go** (от Google).
2. Включите форматирование при сохранении (Settings → “Format on Save”).
3. Создайте tasks.json (Terminal → Configure Tasks) для быстрого запуска:

{

"version": "2.0.0",

"tasks": [

{ "label": "Run myapp", "type": "shell", "command": "go run ./cmd/myapp" },

{ "label": "Build myapp", "type": "shell", "command": "go build -o bin/myapp ./cmd/myapp" }

]

}

**5) Git: первый коммит**

git add .

git commit -m "PZ2: base project layout with logger and /ping"

**6) Проверочные чек-лист/скрины к сдаче**

**Обязательно предоставить:**

1. Скрин структуры проекта:

myapp/

├─ cmd/myapp/main.go

├─ internal/app/app.go

├─ internal/app/handlers/ping.go (если делали Упр. C)

├─ utils/logger.go

├─ utils/httpjson.go (если делали Упр. B)

└─ go.mod

1. Скрин запуска сервера и логов в терминале (после обращений к / и /ping).
2. Скрин/вывод curl -i http://localhost:8080/ping (видно JSON и заголовки).
3. (Если делали Упр. А) — скрин curl -i -H "X-Request-Id: demo-123" ... и заголовка в ответе.

**7) Критерии оценки (10 баллов)**

* **Структура проекта (3 б.)**: есть cmd/, internal/, utils/; main минимален.
* **Работающие ручки (3 б.)**: / (текст) и /ping (JSON).
* **Логирование (2 б.)**: выводится метод/путь/адрес; (доп.) X-Request-Id.
* **Код-стиль (1 б.)**: форматирование, имена пакетов, аккуратные импорты.
* **Git/README (1 б.)**: есть первый коммит и краткий README (что это и как запустить).

**Доп. баллы (+2 суммарно):**

* Единый формат ошибок (utils.WriteErr), маршрут /fail (+1).
* Вынос /ping в отдельный пакет handlers и подключение (+1).

**8) Типовые ошибки и быстрые решения**

* **go run не видит пакет** → Проверьте go.mod и пути импортов (github.com/<user>/myapp/... должны совпадать с go mod init).
* **Порт занят** → Поменяйте порт (в ListenAndServe(":8081", ...)) или завершите процесс, держащий :8080.
* **Кодировка/кракозябры в терминале Windows** → Запускайте PowerShell, шрифт с UTF-8; можно добавить chcp 65001 перед запуском.
* **Не печатаются логи** → Убедитесь, что вызываете utils.LogRequest(r) в каждом хендлере.

**9) Что дальше (на следующее ПЗ)**

* Разнести хендлеры по своим файлам в internal/app/handlers.
* Добавить /time и /echo (POST JSON) с тем же шаблоном логирования.
* Сконфигурировать порт через переменную окружения PORT.

**Визуальная «карта» слоёв (для разговора на занятии)**

* **Вход**: cmd/<app>/main.go — создаёт и запускает приложение. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout))
* **Приложение**: internal/app/... — HTTP-сервер, конфигурация, бизнес-сервисы (приватные пакеты). ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout))
* **Публичные библиотеки** (если появятся): pkg/... — можно переиспользовать в других проектах. ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout))
* **Контракты/артефакты**: api/, configs/, deployments/, scripts/, docs/ и т.д. ([GitHub](https://github.com/golang-standards/project-layout))

**Подготовка отчёта**

* Репозиторий с «минималкой» директорий и рабочим main.go.
* README.md (1–2 абзаца: что это и как запустить).
* Список из 5 артефактов и папок, куда бы вы их положили и **почему** (мини-эссе на 5–10 предложений) — проверяем понимание структуры.

**Вопросы на защиту**

1. Чем internal/ отличается от pkg/ и когда использовать каждый из них? ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout))
2. Почему «много пакетов» может быть вредно, и как этого избегать? (намёк: циклы импорта) ([GitHub](https://github.com/go-standard/project-layout?utm_source=chatgpt.com" \o "go-standard/project-layout))
3. Зачем хранить API-контракты в api/, а не вместе с кодом хендлеров? ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout))
4. Почему src/ в корне — плохая идея для модульных проектов? ([GitHub](https://github.com/golang-standards/project-layout" \o "GitHub - golang-standards/project-layout: Standard Go Project Layout))