**1. Веб-приложение** — программа, которая работает на сервере и предоставляет возможность пользователям взаимодействовать с ней через браузер. В отличие от десктопных приложений, веб-приложения доступны через интернет, и пользователь может использовать их без установки на компьютер.

**Клиент-серверная архитектура:**

Архитектура приложения определяет, как компоненты приложения организованы и взаимодействуют друг с другом. Разные виды архитектуры используются в зависимости от типа приложения, его сложности и требований.

Веб-приложения работают по принципу "клиент-сервер".

**Клиент-серверная** базовая архитектура для большинства веб-приложений, где приложение разделено на две части: клиент (обычно веб-браузер или мобильное приложение) и сервер. Клиент отправляет запросы на сервер, который обрабатывает их и возвращает ответы.

Клиент — это программа, через которую пользователь взаимодействует с веб-приложением (чаще всего это браузер).

Сервер — это компьютер или программа, которая обрабатывает запросы клиентов, выполняет операции и возвращает результат.

**Монолитная** архитектура: вся функциональность разрабатывается и развертывается как единое целое. Все модули (например, бизнес-логика, база данных, пользовательский интерфейс) тесно связаны. Может использоваться для разработки корпоративных приложений, где код всей системы находится в одном большом проекте. Проще в разработке и развертывании на ранних стадиях, Легче отлаживать и тестировать, поскольку все компоненты находятся в одном месте. Сложно масштабировать, так как любые изменения требуют перекомпиляции и развертывания всего приложения, со временем становится трудно поддерживаемым из-за своего размера.

**Микросервисная** архитектура: приложение разделено на несколько независимых сервисов, каждый из которых выполняет определённую задачу. Каждый сервис может разрабатываться, развертываться и масштабироваться отдельно. Можно работать над различными сервисами без конфликта. Требует продуманного управления межсервисной коммуникацией, надежностью и безопасностью.

Архитектура с разделением на слои (**многослойная** архитектура): приложение делится на несколько уровней (слоев), каждый из которых выполняет отдельную функцию. Чаще всего это слои: пользовательского интерфейса (UI), бизнес-логики и базы данных. Каждый слой отвечает за свою конкретную задачу. Слои легче поддерживать и модифицировать: изменения в одном не требуют изменений в других.

**Сервис-ориентированная** архитектура (SOA): основана на предоставлении отдельных сервисов, которые могут взаимодействовать друг с другом через общие интерфейсы, такие как веб-сервисы. Сервисы обычно являются независимыми приложениями, которые могут использоваться несколькими различными приложениями, приложения взаимодействуют через веб-сервисы, например, используя SOAP или REST

**SOAP** (Simple Object Access Protocol — протокол доступа к объектам) протокол, по которому веб-сервисы взаимодействуют друг с другом или с клиентами.

**REST** (Representational state transfer) – архитектурный стиль взаимодействия компонентов распределённого приложения в сети

**Event-Driven** архитектура (Архитектура на основе событий): взаимодействие между компонентами или сервисами осуществляется через события. Когда один компонент завершает свою задачу, он создает событие, которое передается другим компонентам, чтобы они могли реагировать на него. В e-commerce приложениях: события, такие как создание заказа или регистрация пользователя, могут вызвать другие процессы, такие как отправка уведомлений или обновление инвентаря.

**Одностраничные приложения** (SPA — Single Page Application) загружает одну HTML-страницу и динамически обновляет её содержимое по мере взаимодействия пользователя с приложением. При этом страницы не перезагружаются, и взаимодействие с сервером происходит через AJAX-запросы (Приложения на React, Angular, Vue.js)

Основные компоненты веб-приложения:

Фронтенд — часть приложения, которая взаимодействует с пользователем (HTML, CSS, JavaScript).

Бэкенд — серверная часть, которая выполняет логику и обрабатывает запросы.

База данных — место, где хранятся данные (например, MySQL, PostgreSQL).

**2. Протокол HTTP**

HTTP (HyperText Transfer Protocol) — это протокол передачи данных, используемый для обмена информацией между клиентом и сервером. Он работает по принципу запрос-ответ. Когда клиент (браузер) делает запрос к серверу, сервер возвращает ответ в виде HTML, JSON или других данных.

Методы HTTP

GET - запрашивает данные с сервера (например, загрузка страницы).

POST - отправляет данные на сервер (например, отправка формы).

PUT - обновляет существующие данные на сервере.

DELETE - удаляет данные на сервере.

Пример запроса GET: пользователь вводит URL в браузере, браузер отправляет запрос GET на сервер, чтобы загрузить страницу.

**3. Веб-сервер**

программа, которая принимает запросы от клиентов и возвращает им ответы.

Пример работы:

Клиент отправляет HTTP-запрос на веб-сервер.

Сервер обрабатывает запрос и, если нужно, взаимодействует с базой данных.

Сервер возвращает ответ клиенту (HTML-страницу или JSON-данные).

Пример простого веб-сервера на Go:

package main

import (

"fmt"

"net/http"

)

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

fmt.Fprintf(w, "Кто прочитал и выполнил – молодец, скиньте скриншот, чтобы получить плюсик))): %s\n", r.URL.Path)

}

func main() {

http.HandleFunc("/", handler)

fmt.Println("Starting server on :8080")

http.ListenAndServe(":8080", nil)

}

Этот код создает веб-сервер, который обрабатывает запросы и возвращает текстовый ответ. Сервер будет доступен по адресу http://localhost:8080.

**4. Статические и динамические веб-страницы**

Статические страницы — это веб-страницы, которые не изменяются для разных пользователей (например, простая HTML-страница без изменения содержимого).

Динамические страницы — это страницы, содержание которых может изменяться в зависимости от запросов пользователя или других данных (например, страница с результатами поиска или пользовательский профиль).

**5. Маршрутизация запросов**

Маршрутизация (routing) — это процесс, с помощью которого веб-сервер определяет, какой код или функция должны быть вызваны в ответ на запрос клиента.

package main  
import (  
 "fmt"  
 "net/http"  
)  
func homeHandler(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Добро пожаловать на главную страницу!")

}  
func aboutHandler(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Это страница о нас.")

}  
func main() {  
 http.HandleFunc("/", homeHandler)  
 http.HandleFunc("/about", aboutHandler)  
 fmt.Println("Сервер запущен на порту :8080")  
 http.ListenAndServe(":8080", nil)

}

**Работа с параметрами запросов**

Вы можете извлекать параметры из URL с помощью r.URL.Query(). Например, если вы хотите получить имя пользователя из запроса:

func queryHandler(w http.ResponseWriter, r \*http.Request) {  
 name := r.URL.Query().Get("name")  
 fmt.Fprintf(w, "Привет, %s!", name)  
}

**Использование сторонних маршрутизаторов**

Для более сложной маршрутизации в Go можно использовать сторонние маршрутизаторы, такие как **Gorilla Mux**. Вот пример использования для маршрутизации:

package main  
import (  
 "fmt"  
 "github.com/gorilla/mux"  
 "net/http"  
)  
func homeHandler(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Добро пожаловать на главную страницу!")  
}  
func aboutHandler(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Это страница о нас.")  
}  
func main() {  
 r := mux.NewRouter()  
 r.HandleFunc("/", homeHandler)  
 r.HandleFunc("/about", aboutHandler)  
 http.ListenAndServe(":8080", r)  
}

**6. Рендеринг HTML-шаблонов**

Go поддерживает работу с HTML-шаблонами через пакет html/template. Это позволяет генерировать HTML-страницы динамически

package main

import (

"html/template"

"net/http"

)

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

tmpl := template.Must(template.ParseFiles("animals.html"))

tmpl.Execute(w, nil)

}

func main() {

http.HandleFunc("/", handler)

http.ListenAndServe(":8080", nil)

}

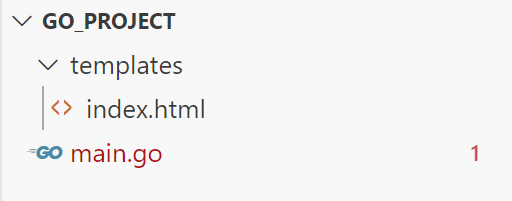
Структура проекта:

go\_project/

│

├── main.go

├── templates/

│ └── animals.html   


**7. Введение в REST API**

REST (Representational State Transfer) — это архитектурный стиль, который используется для создания веб-сервисов. Он основывается на стандарте HTTP и позволяет взаимодействовать с веб-сервисами с помощью стандартных методов HTTP (GET, POST, PUT, DELETE).

Пример работы REST API с Зоопарком:

* + GET /animals - возвращает список всех животных.
  + POST /animals - создает новое животное.
  + GET /animals/{id} – возвращает информацию о животном по ID.
  + PUT /animals/{id} - обновляет информацию о животном.
  + DELETE /animals/{id} - удаляет животное.

package main  
import (  
 "encoding/json"  
 "net/http"  
)

var animals = map[int]string{  
 1: "Лев",  
 2: "Тигр",  
}

func getAnimals(w http.ResponseWriter, r \*http.Request) {  
 json.NewEncoder(w).Encode(animals)

}

func main() {  
 http.HandleFunc("/animals", getAnimals)  
 http.ListenAndServe(":8080", nil)

}

Этот код создает простой REST API, который возвращает список животных в формате JSON.

**8. Аутентификация и авторизация пользователей**

Аутентификация — процесс проверки личности пользователя, а авторизация — предоставление разрешений на доступ к различным частям приложения.

Существует несколько видов пользователей, например, обычный пользователь и администратор, что позволяет контролировать доступ к функциональности.

package main

import (  
 "fmt"  
 "net/http"

)

var users = map[string]string{  
 "user": "password123",  
 "admin": "adminpass",

}

func loginHandler(w http.ResponseWriter, r \*http.Request) {  
 username := r.FormValue("username")  
 password := r.FormValue("password")

if pass, ok := users[username]; ok && pass == password {

fmt.Fprintf(w, "Добро пожаловать, %s!", username)

} else {  
 http.Error(w, "Неверные учетные данные", http.StatusUnauthorized)

}

}

func main() {  
 http.HandleFunc("/login", loginHandler)  
 http.ListenAndServe(":8080", nil)

}

**9. Обработка запросов пользователя**

Обработка запросов происходит через методы HTTP (GET, POST и др.), позволяя взаимодействовать с веб-приложением.

Это включает отправку форм, получение данных и выполнение других операций.

package main  
import (  
 "fmt"  
 "net/http"

)

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

if r.Method == http.MethodPost {  
 name := r.FormValue("name")  
 fmt.Fprintf(w, "Привет, %s!", name)  
 } else {  
 http.ServeFile(w, r, "form.html")  
 }  
}  
func main() {  
 http.HandleFunc("/form", formHandler)  
 http.ListenAndServe(":8080", nil)

}