

**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Новосибирский государственный университет экономики и управления «НИНХ»**

**(ФГБОУ ВО «НГУЭУ», НГУЭУ)**

Кафедра информационных технологий

**ОТЧЕТ О ПРОХОЖДЕНИИ ПРАКТИКИ**

Направление: 09.03.02 Информационные системы и технологии

Вид практики: учебная практика

Тип практики: Ознакомительная практика

Место прохождения практики: НГУЭУ, кафедра информационных технологий, 630099, г. Новосибирск, ул. Каменская, 52/1

Сроки прохождения практики: с 23 июня 2025 г. по 5 июля 2025 г.

Выполнил:

Студент(ка) гр. ИС401 5 июля 2025 г. О.С. Чернышов

Руководитель практики

от профильной организации

Заведующий кафедрой ИТ 5 июля 2025 г. А.И. Пестунов

Руководитель практики

от университета

Кандидат технических наук, доцент 5 июля 2025 г. А.Ю. Пигарев

Новосибирск 2025



**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Новосибирский государственный университет экономики и управления «НИНХ»**

**(ФГБОУ ВО «НГУЭУ», НГУЭУ)**

Кафедра информационных технологий

**ЗАДАНИЕ НА ПРАКТИКУ**

Направление: 09.03.02 Информационные системы и технологии

Вид практики: учебная практика

Тип практики: Ознакомительная практика

Место практики: НГУЭУ, кафедра информационных технологий, 630099, г. Новосибирск, ул. Каменская, 52/1

Сроки прохождения практики: с 23 июня 2025 г. по 5 июля 2025 г.

Выдано студенту 1 курса ИС401 группы

Чернышову Олегу Сергеевичу

Индивидуальное задание на практику, содержание, планируемые результаты: целью практики является освоение и применение современных технологий разработки веб-приложений на примере реализации программного приложения «Спортивные клубы». Приложение должно предоставлять пользователю возможность просмотра, поиска, спортивных клубах через веб-интерфейс.

Рабочий график (план) проведения практики

|  |  |
| --- | --- |
| Этапы практики | Период |
| Ознакомление с темой. | 23.06.25 – 24.06.25 |
| Составление плана работы, цели, задачи. | 24.06.25 – 27.06.25 |
| Реализация информационной системы. | 27.06.25 – 04.07.25 |
| Изучение полученных данных, составление и подготовка отчета о практике к защите. | 04.06.25 – 04.07.25 |

Руководитель практики

от университета

Кандидат технических наук, доцент 23 июня 2025 г. А.Ю. Пигарев

Задание согласовано

Руководитель практики от профильной организации

Заведующий кафедрой ИТ

А.И. Пестунов 23.06.2025

Задание получено

Чернышов Олег Сергеевич 23.06.2025



**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Новосибирский государственный университет экономики и управления «НИНХ»**

**(ФГБОУ ВО «НГУЭУ», НГУЭУ)**

Кафедра информационных технологий

**ДНЕВНИК ПРОХОЖДЕНИЯ ПРАКТИКИ**

Направление: 09.03.02 Информационные системы и технологии

Вид практики: учебная практика

Тип практики: Ознакомительная практика

Место практики: НГУЭУ, кафедра информационных технологий, 630099, г. Новосибирск, ул. Каменская, 52/1

Сроки прохождения практики: с 23 июня 2025 г. по 5 июля 2025 г.

Студент(ка) 1 курса, ИС401 группы

Чернышов Олег Сергеевич

Работал: НГУЭУ

|  |  |  |
| --- | --- | --- |
| Дата | Краткое описание видов работ | Отметка о выполнении |
| 23.07.2025 | Ознакомление с инструкциями организации по правилам противопожарной безопасности, правилам охраны труда, техники безопасности; требованиями по соблюдению санитарно-эпидемиологических правил и гигиенических нормативов; планами эвакуации при возникновении пожара; правилами внутреннего трудового распорядка | ✓ |
| 24.07.2025 | Сбор и анализ научных источников | ✓ |
| 25.07.2025 | Продумывание архитектуры проекта и структуры базы данных | ✓ |
| 26.07.2025 | Разработка серверной части (Backend) | ✓ |
| 02.07.2025 | Разработка клиентской части (Frontend) | ✓ |
| 03.07.2025 | Составление и оформление отчета по практике | ✓ |
| 04.07.2025 | Тестирование, демонстрация работы приложения | ✓ |
| 05.07.2025 | Защита практики | ✓ |

Студент(ка)

Чернышов Олег Сергеевич 05.07.2025

Содержание и объем выполненных работ подтверждаю.

Руководитель практики от профильной организации

А.И. Пестунов 05.07.2025

Работы выполнены в установленные сроки, содержание практики соответствует индивидуальному заданию.

Руководитель практики

от университета

Кандидат технических наук, доцент 5 июля 2025 г. А.Ю. Пигарев

СОДЕРЖАНИЕ

[**СОДЕРЖАНИЕ 6**](#_Toc203407560)

[**Глава 1. Теоретическая часть 9**](#_Toc203407561)

[**Архитектура приложения 10**](#_Toc203407562)

[**Структура базы данных. 11**](#_Toc203407563)

[**Подходы к разработке. 12**](#_Toc203407564)

[**Практическая часть. 14**](#_Toc203407565)

[**Backend разработка. 14**](#_Toc203407566)

[**Frontend разработка. 16**](#_Toc203407567)

[**Тестирование и результаты 18**](#_Toc203407568)

[**Подготовка к началу тестов. 18**](#_Toc203407569)

[**Тестирование API (Backend) 18**](#_Toc203407570)

[**Тестирование frontend (Javascript + UI) 19**](#_Toc203407571)

[**Тестирование интерфейса (CSS + UX) 20**](#_Toc203407572)

[**Демонстрация работы приложения. 21**](#_Toc203407573)

[**ЗАКЛЮЧЕНИЕ 22**](#_Toc203407574)

[**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 23**](#_Toc203407575)

[**ПРИЛОЖЕНИЕ А 24**](#_Toc203407576)

[**ПРИЛОЖЕНИЕ Б 31**](#_Toc203407577)

[**ПРИЛОЖЕНИЕ В 32**](#_Toc203407578)

**ВВЕДЕНИЕ**

Веб-разработка сегодня является важнейшим инструментом для автоматизации бизнес-процессов, удобного доступа к информации и повышения качества взаимодействия с пользователями. Благодаря постоянному развитию технологий появляются новые решения, позволяющие создавать производительные, масштабируемые и удобные приложения. Особую значимость приобретает разработка информационных систем, которые обеспечивают наглядное представление данных и их обработку с применением современных методов.

Целью данной работы является создание веб-приложения для автосервиса, в котором серверная часть реализована на языке Go (Golang), для хранения данных используется СУБД PostgreSQL, а клиентская часть построена на HTML, CSS и JavaScript. Приложение позволяет просматривать и искать информацию об автомобилях, предоставляя пользователям удобный интерфейс для работы с данными.

В процессе разработки были поставлены следующие задачи:

1. Спроектировать структуру базы данных, соответствующую предметной области, с таблицами brands, models и cars, учитывая нормализацию и связи между ними.

2. Разработать серверную часть на языке Go.

3. Реализовать frontend с интуитивно понятным интерфейсом.

4. Обеспечить динамическую загрузку моделей автомобилей в зависимости от выбранной марки.

5. Обеспечить интеграцию frontend и backend, организовав обмен данными через HTTP-запросы в формате JSON.

6. Реализовать валидацию пользовательского ввода и обработку ошибок.

7. Настроить отображение результатов поиска в виде таблицы.

В результате был реализован полный цикл разработки веб-приложения: от проектирования базы данных до создания пользовательского интерфейса и тестирования. Приложение построено по клиент-серверной архитектуре, где сервер отвечает за логику и работу с данными, а клиентская часть — за отображение информации и взаимодействие с пользователем. Такой подход обеспечил модульность, гибкость и возможность дальнейшего расширения функционала.

Выбор технологий обусловлен их преимуществами: Go обеспечивает высокую производительность и поддержку параллельных вычислений, PostgreSQL предлагает надежные механизмы хранения и обработки данных, а связка HTML/CSS/JavaScript позволяет быстро разрабатывать кроссплатформенные интерфейсы без использования сторонних фреймворков.

В результате выполнения практики было разработано полноценное веб-приложение, которое демонстрирует работу с базой данных, динамическим интерфейсом и обработкой пользовательских запросов. Проект может быть расширен дополнительными функциями, такими как сортировка результатов, пагинация или интеграция с внешними API для получения актуальных данных об автомобилях.

# Глава 1. Теоретическая часть

Веб-приложение для поиска автомобилей разработано с использованием современных технологий, которые обеспечивают надежность, производительность и удобство взаимодействия с пользователем. Проект сочетает в себе серверную часть на языке Go (Golang), frontend на HTML, CSS и JavaScript, а также базу данных PostgreSQL для хранения информации. Каждый компонент технологического стека был выбран исходя из его преимуществ и соответствия задачам проекта.

Основу серверной части составляет язык программирования **Go (Golang)**, который был выбран благодаря своей простоте, высокой производительности и эффективной работе с многопоточными задачами. Go обеспечивает быстрое выполнение запросов и минимальные накладные расходы, что особенно важно для веб-приложений. В рамках проекта использовались возможности и сильные стороны библиотек Go (Golang).

Стандартная библиотека Go (Golang) net/http позволила легко настрить обработчики для эндпоинтов главной страницы и поиска автомобилей, пакет html/template дал возможность генерировать HTML-страницы, обеспечивая безопасное и удобное встраивание данных в шаблоны с возможностью динамического форматирование страницы на основе данных, полученных из базы. Для подключения к PostgreSQL и выполнения SQL-запросов применялась библиотека pgx. Она предоставляет удобный интерфейс для работы с базой данных, включая поддержку контекста для отмены длительных операций.

Для реализации клиентской части использован статический веб-интерфейс, построенный на базе HTML, CSS и JavaScript. Это решение позволило создать отзывчивый интерфейс с возможностью обновления данных в реальном времени, избежав зависимости от внешних фреймворков и сохранив высокую производительность.

## 1.1. Архитектура приложения

Разработанное веб-приложение для поиска автомобилей построено по клиент-серверной архитектуре. Серверная часть обрабатывает HTTP-запросы, взаимодействует с базой данных и возвращает данные в формате JSON, а клиентская часть динамически отображает их пользователю.

Ключевые модули:

* main.go — точка входа, настраивает маршруты (/, /search), запускает сервер и управляет подключением к БД.
* body.html и header.html — шаблоны страниц, формируемые сервером с использованием html/template.
* style.css — стилизация интерфейса (форма поиска, таблица результатов).
* models\_label.js — динамическая подгрузка моделей автомобилей в зависимости от выбранной марки.
* search.js — обработка формы поиска, валидация данных, отправка запросов на сервер и отображение результатов.

Особенности архитектуры:

* Четкое разделение на frontend и backend.
* Использование шаблонов Go для серверного рендеринга.
* Асинхронная загрузка данных через Fetch API.
* Подготовленные SQL-запросы для безопасного взаимодействия с БД.

Архитектура обеспечивает производительность, безопасность и удобство масштабирования.

## 1.2. Структура базы данных.

База данных приложения спроектирована с учетом нормализации и эффективного хранения информации об автомобилях. Она состоит из трёх связанных таблиц, которые обеспечивают целостность данных и удобство выборки. База данных состоит из трёх связанных таблиц: brands, models и cars.

Таблица brands предназначена для хранения информации о производителях автомобилей. Она содержит в себе следующие поля:

* id (INT, PRIMARY KEY) – уникальный идентификатор поля
* name (VARCHAR) – название марки автомобиля, заданное как уникальное и не допускающее значения NULL.

Таблица models предназначена для хранения информации о моделях конкретной марки и включает в себя следующие поля:

* Id (INT, PRIMARY KEY) – уникальный идентификатор модели.
* name (VARCHAR) — название модели
* brand\_id (INT, FOREIGN KEY) — ссылка на марку из таблицы Brands

Таблица cars представляет из себя хранилище конкретных автомобилей с их характеристиками. Она включает в себя следующие поля:

* id (INT, PRIMARY KEY) — уникальный идентификатор автомобиля.
* model\_id (INT, FOREIGN KEY) — ссылка на модель из таблицы Models.
* year (INT) — год выпуска.
* price (INT) — цена в тыс. руб.

Между таблицами сформирована связь «один ко многим», поскольку один производитель может выпустить несколько различных марок автомобилей. Каждая марка автомобиля, в свою очередь, может иметь разные цены в зависимости от продавца (система с продавцом включена в планы масштабирования проекта). Наличие внешних ключей brand\_id и model\_id обеспечивает ссылочную целостность и предотвращает отображение некорректных записей.

Для демонстрации работы приложения подготовлены тестовые данные, включающие информацию о двадцати пяти моделей автомобилей от разных производителей. Эти данные были созданы в базе данных с помощью SQL-скрипта demo.sql, который позволил получить рабочий набор данных для просмотра списков сразу после запуска сервера.

## 1.3 Подходы к разработке.

В процессе разработки был принят подход, сочетающий модульность и минимализм. Серверная часть на Go (Golang) организована вокруг единой точки входа main.go, где сосредоточена логика маршрутизации и работы с базой данных, что упрощает поддержку и расширение кода. Для взаимодействия с PostgreSQL использовалась библиотека pgx, обеспечивающая безопасность запросов за счёт параметризованных SQL-выражений, а данные передавались в формате JSON, что сделало API универсальным и легко интегрируемым с клиентской частью.

На frontend-части применялся принцип постепенного улучшения: базовая функциональность работала даже без JavaScript (например, выбор марки и модели), а динамическое поведение, такое как валидация формы и асинхронная загрузка результатов, добавлялось с помощью компактных JavaScript-модулей. Это позволило сохранить интерфейс отзывчивым и удобным для пользователя. Для стилизации использовался чистый CSS без избыточных фреймворков, что обеспечило быструю загрузку страниц и простую кастомизацию.

Особое внимание уделялось обработке ошибок и обратной связи с пользователем. Клиентская валидация форм предотвращала некорректные запросы, а сервер возвращал понятные сообщения об ошибках в формате JSON. Логирование ключевых операций (например, запросов к базе данных) помогало оперативно выявлять проблемы. Такой подход минимизировал количество сбоев и улучшил пользовательский опыт.

В целом, проект разрабатывался с упором на простоту и эффективность. Отказ от избыточных зависимостей в пользу нативных технологий (Go, Vanilla JS) и продуманная структура базы данных позволили создать легковесное, но функциональное приложение, которое легко масштабировать и адаптировать под новые требования.

# Глава 2. Практическая часть.

## 2.1. Backend разработка.

Backend-часть приложения для поиска автомобилей построена на языке Go с использованием фреймворка для работы с базами данных pgx и стандартной библиотеки HTTP. Сервер обеспечивает загрузку данных о марках и моделях автомобилей, а также предоставляет функционал для поиска автомобилей по заданным критериям. Взаимодействие с базой данных организовано через контексты, что позволяет контролировать время выполнения запросов и избегать зависаний.

Основная функция MainForm отвечает за отображение главной страницы. Она загружает данные о марках и моделях автомобилей из базы данных, используя функции getBrands и getModels, а затем передаёт их в HTML-шаблон. Для удобства работы с данными в шаблоне используется функция json, которая преобразует структуры Go в JSON-строку. Это позволяет динамически обновлять список моделей на клиенте в зависимости от выбранной марки.

Функция SearchCars обрабатывает запросы на поиск автомобилей. Она принимает параметры из URL (марка, модель, год выпуска и цена), формирует SQL-запрос с учётом переданных фильтров и возвращает результаты в формате JSON. Для построения запроса используется гибкая система условий, которая добавляет фильтры только при наличии соответствующих параметров. Это делает поиск удобным и адаптивным под нужды пользователя.

Работа с базой данных вынесена в отдельные функции. getBrands загружает список всех марок автомобилей, а getModels — список всех моделей с привязкой к маркам. Функция getCars выполняет основной поисковый запрос, объединяя таблицы автомобилей, моделей и марок для получения полной информации. Каждая из этих функций обрабатывает возможные ошибки и логирует ключевые этапы работы, что упрощает отладку приложения.

Для подключения к базе данных используется функция connectDB, которая читает параметры подключения из переменных окружения. Это обеспечивает безопасность и гибкость при развёртывании приложения в разных средах. Контексты с таймаутом позволяют избежать долгих ожиданий при проблемах с сетью или базой данных.

Сервер также обслуживает статические файлы (CSS и JavaScript), которые необходимы для работы frontend-части. Это позволяет клиенту динамически обновлять интерфейс без перезагрузки страницы. Взаимодействие между frontend и backend построено на REST-подходе, где клиент отправляет параметры поиска через URL, а сервер возвращает данные в формате JSON.

В целом, backend-часть приложения обеспечивает надёжное и быстрое взаимодействие с базой данных, гибкий поиск по критериям и удобную интеграцию с frontend-частью. Использование современных практик, таких как контексты и работа с переменными окружения, делает код устойчивым к ошибкам и легко масштабируемым.

## 2.2. Frontend разработка.

Frontend-часть приложения для поиска автомобилей реализована с использованием чистого HTML, CSS и JavaScript, что обеспечивает простоту и отзывчивость интерфейса без необходимости в дополнительных фреймворках. Основной акцент сделан на удобстве пользователя: форма поиска интерактивна, а результаты отображаются динамически без перезагрузки страницы.

Главная страница состоит из формы поиска, включающей выпадающие списки марок и моделей, поля для ввода года выпуска и максимальной цены. При выборе марки в списке моделей автоматически подгружаются только соответствующие варианты благодаря функции updateModels из файла models\_label.js. Данные о моделях передаются из backend-части в виде JSON, что позволяет быстро фильтровать их на стороне клиента без дополнительных запросов к серверу.

Обработка формы поиска происходит в файле search.js. При отправке формы данные валидируются: проверяется, что хотя бы одно поле заполнено, а год и цена являются числами. Если проверка пройдена, формируется URL с параметрами запроса, и с помощью Fetch API отправляется GET-запрос на сервер. Полученные результаты отображаются в виде таблицы с помощью функции renderResults, которая динамически генерирует HTML-разметку. Если автомобили не найдены, пользователь видит соответствующее сообщение.

Стилизация интерфейса выполнена в файле style.css. Дизайн выдержан в минималистичном и современном стиле с акцентом на удобство восприятия. Таблица результатов поиска адаптирована под разные размеры экранов, а интерактивные элементы (кнопки, поля ввода) имеют визуальную обратную связь при наведении и фокусе. Сообщения об ошибках и уведомления выделяются цветом, что делает интерфейс более дружелюбным.

Таким образом, frontend-часть приложения обеспечивает плавное взаимодействие пользователя с системой. Динамическая подгрузка данных, мгновенный вывод результатов и продуманный дизайн делают процесс поиска автомобилей интуитивно понятным и комфортным.

# Глава 3. Тестирование и результаты

## 3.1. Подготовка к началу тестов.

Последний этап разработки веб-приложения "Автосалон" включал в себя разносторонний обзор его функциональности, устойчивости к ошибкам и качества пользовательского интерфейса. Цель тестирования заключалась в обнаружении различных ошибок и нестандартного поведения, проверка работоспособности всех функций и установление надёжного отношения между клинтской и серверной частью.

Этапы тестирования разделились на три этапа: сначала шла проверка отдельных функций приложения, затем рассматривалось их роль в общей картине программы, а затем стабильность и производительность всего кода в целом. Результаты тестирования сохранены для дальнейшей работы и проектирования.

Перед тесированием были проведены следующие действия:

- Установлено необходимое ПО для работы: PostgreSQL, Go, браузер edge и firefox.

- Настроены таблицы brands, models и cars в базе данных.

- Открыт локальный сервер через команду go run main.go используя порт 8080.

- Настроено подключение к базе данных через драйвер pgx. Установлена корректная работа

## 3.2. Тестирование API (Backend)

* Тест 1: Получение списка марок и моделей
  + Действие: Запрос к **/** (главная страница).
  + Ожидаемый результат: возвращается HTML-страница с заполненным списком марок и скрытым списком моделей.
  + Результат: Успешно. Данные загружаются, модели отображаются только после выбора марки.
* Тест 2: Поиск автомобилей без фильтров
  + Действие: GET-запрос /search без параметров.
  + Ожидаемый результат: возвращается список из всех автомобили, содержащихся в базе данных.
  + Результат: Успешно. Сервер возвращает полный список.
* Тест 3: Поиск по марке (например, BMW)
  + Действие: GET-запрос /search?brand=1 (ID BMW).
  + Ожидаемый результат: возвращаются только автомобили марки BMW.
  + Результат: Успешно. В выборке только BMW.
* Тест 4: Поиск по несуществующей марке
  + Действие: GET-запрос /search?brand=999.
  + Ожидаемый результат: пустой массив (автомобили не найдены).
  + Результат: Успешно. Сервер возвращает [].
* Тест 5: Поиск с некорректными параметрами (год = "abc")
  + Действие: GET-запрос /search?year=abc.
  + Ожидаемый результат: ошибка 400 (неверный формат данных).
  + Результат: Успешно. Сервер возвращает 400 Bad Request.

## 3.3. Тестирование frontend (Javascript + UI)

* Тест 6: Загрузка и отображение моделей при выборе марки
  + Действие: Выбор марки в <select id="brand">.
  + Ожидаемый результат: Список моделей обновляется, показывая только модели выбранной марки.
  + Результат: Успешно. Модели фильтруются корректно.
* Тест 7: Отправка формы без параметров
  + Действие: Нажать кнопку "Поиск" без ввода данных.
  + Ожидаемый результат: Появляется сообщение "Укажите хотя бы один критерий поиска".
  + Результат: Успешно. Валидация работает.
* Тест 8: Ввод букв в поле "Год"
  + Действие: Ввод abc в поле года.
  + Ожидаемый результат: Сообщение "Год должен быть числом".
  + Результат: Успешно. Ошибка отображается.
* Тест 9: Поиск автомобиля с корректными данными
  + Действие: Выбор марки, модели, ввод года и цены, затем нажать кнопку «Поиск».
  + Ожидаемый результат: Таблица с результатами поиска.
  + Результат: Успешно. Данные отображаются в таблице.
* Тест 10: Поиск несуществующего автомобиля
  + Действие: Ввод заведомо неверных параметров (например, год 3000).
  + Ожидаемый результат: Сообщение "Автомобили не найдены".
  + Результат: Успешно. Выводится уведомление.

## 3.4. Тестирование интерфейса (CSS + UX)

* Тест 11: Адаптивность таблицы результатов
  + Действие: Изменение ширины экрана.
  + Ожидаемый результат: Таблица не выходит за границы контейнера.
  + Результат: Успешно. Таблица адаптируется.
* Тест 12: Визуальная обратная связь при наведении
  + Действие: Наведение на строки таблицы и кнопку поиска.
  + Ожидаемый результат: Элементы подсвечиваются.
  + Результат: Успешно. CSS-эффекты работают.
* Тест 13: Отображение ошибок и сообщений
  + Действие: Проверка стилей ошибок (class="error") и уведомлений (class="message").
  + Ожидаемый результат: Красный фон для ошибок, зелёный — для сообщений.
  + Результат: Успешно. Стили применяются корректно.

## 3.5. Демонстрация работы приложения.

Для наглядности функционала проекта было подготовлены видео и gif-анимация, содержание которых демонстрировало основные возможности продукта. Видео содержит следующие этапы:

* Открытие страницы и отображение чистого, минималистичного дизайна с заголовком "Поиск автомобилей"
* Выполнение поиска по марке и бренду
* Попытка ввода некорректных данных в поля "год" и "цена"

Видеозапись размещена в репозитории проекта на GitHub, что позволяет без проблем ознакомиться с работой приложения без необходимости его локального запуска.

Тестирование подтвердило стабильную и корректную работу всех компонентов приложения: backend успешно обрабатывает запросы, предоставляя актуальные данные, frontend обеспечивает интуитивно понятный интерфейс с мгновенной реакцией на действия пользователя, а система валидации исключает ошибки ввода. Интерфейс демонстрирует отзывчивость на различных устройствах, а продуманные визуальные подсказки делают взаимодействие с приложением комфортным. Несмотря на безупречные результаты, для дальнейшего развития проекта стоит рассмотреть возможность добавления пагинации и оптимизации запросов, что особенно актуально при работе с большими объемами данных. В целом, приложение полностью готово к использованию, сочетая в себе надежность, удобство и приятный дизайн.

# ЗАКЛЮЧЕНИЕ

В ходе работы над проектом был создан удобный и функциональный инструмент для поиска автомобилей, сочетающий в себе простоту интерфейса и надежность backend-решений. Использование современных технологий, таких как Go для серверной части и чистого JavaScript для динамического взаимодействия с пользователем, позволило добиться высокой производительности и отзывчивости системы.

Особое внимание уделялось удобству работы с данными — благодаря продуманной структуре базы данных и строгой нормализации, приложение эффективно обрабатывает запросы и обеспечивает целостность информации. Четкое разделение логики между клиентской и серверной частями, а также продуманная валидация ввода сделали систему устойчивой к ошибкам и простой в использовании.

В результате получилось компактное, но мощное веб-приложение, готовое к масштабированию и доработкам. Оно демонстрирует, как даже в небольшом проекте можно успешно сочетать чистый код, оптимальную архитектуру и удобный интерфейс. Этот опыт подтвердил, что грамотный подход к проектированию и выбору технологий позволяет создавать решения, которые остаются гибкими, производительными и удобными для конечного пользователя.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. The Go Programming Language: Documentation [Электронный ресурс] // Documentation – официальная документация языка программирования Go. — Режим доступа: https://golang.org/doc/ (дата обращения: 25.06.2025). — Текст: электронный.

2. PostgreSQL: Документация по установке и настройке СУБД [Электронный ресурс] // PostgreSQL Official Documentation – информация о настройке локального сервера PostgreSQL, создании баз данных и пользователей, а также выполнении SQL-скриптов. — Режим доступа: https://www.postgresql.org/docs/ (дата обращения: 26.06.2025). — Текст: электронный.

3. HTML, CSS и JavaScript: MDN Web Docs [Электронный ресурс] // MDN Web Docs – справочные материалы по разработке клиентской части интерфейса с использованием стандартных технологий веб-разработки. — Режим доступа: https://developer.mozilla.org/ru/ (дата обращения: 30.06.2025). — Текст: электронный.

4. Golang + PostgreSQL: Пример реализации REST API [Электронный ресурс] // Tutorialedge – статья с примером построения серверной части приложения на основе Go и PostgreSQL с поддержкой CRUD операций и JSON-ответов. — Режим доступа: https://tutorialedge.net/golang/authenticating-golang-rest-api-with-jwts/ (дата обращения: 03.07.2025). — Текст: электронный.

5. JavaScript: Современный учебник по разработке клиентской части веб-приложений [Электронный ресурс] // JavaScript.info – подробное описание возможностей JavaScript, AJAX-запросов и динамического обновления DOM-дерева. — Режим доступа: https://javascript.info/ (дата обращения: 03.07.2025). — Текст: электронный.

# ПРИЛОЖЕНИЕ А

Код файла main.go

package main

import (

    "context"

    "encoding/json"

    "fmt"

    "log"

    "net/http"

    "os"

    "strings"

    "text/template"

    "time"

    "github.com/jackc/pgx/v5"

    "github.com/joho/godotenv"

)

type Brand struct {

    ID   *int*    `json:"id"`

    Name *string* `json:"name"`

}

type Model struct {

    ID      *int*    `json:"id"`

    BrandID *int*    `json:"brand\_id"`

    Name    *string* `json:"name"`

}

type Car struct {

    ID        *int*    `json:"id"`

    BrandName *string* `json:"brand\_name"`

    ModelName *string* `json:"model\_name"`

    Year      *int*    `json:"year"`

    Price     *int*    `json:"price"`

}

func main() {

    err := godotenv.Load()

    if err != nil {

        log.Fatal("Ошибка загрузки .env файла")

    }

    http.Handle("/static/", http.StripPrefix("/static/", http.FileServer(http.Dir("static"))))

    http.HandleFunc("/", MainForm)

    http.HandleFunc("/search", SearchCars)

    fmt.Println("Сервер работает на http://localhost:8080")

    http.ListenAndServe(":8080", nil)

}

func connectDB(*ctx* context.Context) (\*pgx.Conn, *error*) {

    conn, err := pgx.Connect(ctx, os.Getenv("DATABASE\_URL"))

    if err != nil {

        return nil, fmt.Errorf("ошибка подключения к базе данных: %v", err)

    }

    return conn, nil

}

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

    ctx, cancel := context.WithTimeout(r.Context(), 5\*time.Second)

    defer cancel()

    conn, err := connectDB(ctx)

    if err != nil {

        http.Error(w, "Ошибка подключения к БД: "+err.Error(), http.StatusInternalServerError)

        return

    }

    defer conn.Close(ctx)

    brandsRows, err := getBrands(conn, ctx)

    if err != nil {

        http.Error(w, "Ошибка получения марок: "+err.Error(), http.StatusInternalServerError)

        return

    }

    modelsRows, err := getModels(conn, ctx)

    if err != nil {

        http.Error(w, "Ошибка получения моделей: "+err.Error(), http.StatusInternalServerError)

        return

    }

    Data := struct {

        Brands []Brand

        Models []Model

    }{

        Brands: brandsRows,

        Models: modelsRows,

    }

*//JSON*

    var funcMap = template.FuncMap{

        "json": func(*v* interface{}) (*string*, *error*) {

            b, err := json.Marshal(v)

            return *string*(b), err

        },

    }

    tmpl := template.Must(template.New("").Funcs(funcMap).ParseFiles("templates/body.html", "templates/header.html"))

    tmpl.ExecuteTemplate(w, "body", Data)

}

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

    w.Header().Set("Access-Control-Allow-Origin", "\*")

    w.Header().Set("Content-Type", "application/json")

    ctx, cancel := context.WithTimeout(r.Context(), 5\*time.Second)

    defer cancel()

    conn, err := connectDB(ctx)

    if err != nil {

        http.Error(w, "Ошибка подключения к БД: "+err.Error(), http.StatusInternalServerError)

        return

    }

    defer conn.Close(ctx)

    brandID := r.URL.Query().Get("brand")

    modelID := r.URL.Query().Get("model")

    year := r.URL.Query().Get("year")

    price := r.URL.Query().Get("price")

    cars, err := getCars(conn, ctx, brandID, modelID, year, price)

    if err != nil {

        http.Error(w, "Ошибка поиска автомобилей: "+err.Error(), http.StatusInternalServerError)

        return

    }

    json.NewEncoder(w).Encode(cars)

}

func getBrands(*connection* \*pgx.Conn, *ctx* context.Context) ([]Brand, *error*) {

    brandsRows, err := connection.Query(ctx, "SELECT id, name FROM brands")

    if err != nil {

        return nil, fmt.Errorf("ошибка загрузки марок: %w", err)

    }

    defer brandsRows.Close()

    var dataRows []Brand

    for brandsRows.Next() {

        var b Brand

        err := brandsRows.Scan(&b.ID, &b.Name)

        if err != nil {

            return nil, fmt.Errorf("ошибка сканирования марки: %v", err)

        }

        dataRows = append(dataRows, b)

    }

*//логи*

    if err := brandsRows.Err(); err != nil {

        return nil, fmt.Errorf("ошибка при обработке строк: %v", err)

    }

    fmt.Printf("Отладка: загружено марок — %d\n", len(dataRows))

    return dataRows, nil

}

func getModels(*connection* \*pgx.Conn, *ctx* context.Context) ([]Model, *error*) {

    modelsRows, err := connection.Query(ctx, "SELECT id, brand\_id, name FROM models")

    if err != nil {

        return nil, fmt.Errorf("ошибка загрузки моделей: %w", err)

    }

    defer modelsRows.Close()

    var dataRows []Model

    for modelsRows.Next() {

        var m Model

        err := modelsRows.Scan(&m.ID, &m.BrandID, &m.Name)

        if err != nil {

            return nil, fmt.Errorf("ошибка сканирования модели: %v", err)

        }

        dataRows = append(dataRows, m)

    }

*//логи*

    if err := modelsRows.Err(); err != nil {

        return nil, fmt.Errorf("ошибка при обработке строк: %v", err)

    }

    fmt.Printf("Отладка: загружено моделей — %d\n", len(dataRows))

    return dataRows, nil

}

func getCars(*connection* \*pgx.Conn, *ctx* context.Context, *brandID*, *modelID*, *year*, *price* *string*) ([]Car, *error*) {

    log.Printf("Поиск автомобилей: brand=%s, model=%s, year=%s, price=%s", brandID, modelID, year, price)

    query := `

        SELECT c.id, b.name AS brand\_name, m.name AS model\_name, c.year, c.price

        FROM cars c

        JOIN models m ON c.model\_id = m.id

        JOIN brands b ON m.brand\_id = b.id

        WHERE 1=1

    `

    var args []interface{}

    var conditions []*string*

    if brandID != "" {

        args = append(args, brandID)

        conditions = append(conditions, fmt.Sprintf("AND b.id = $%d", len(args)))

    }

    if modelID != "" {

        args = append(args, modelID)

        conditions = append(conditions, fmt.Sprintf("AND m.id = $%d", len(args)))

    }

    if year != "" {

        args = append(args, year)

        conditions = append(conditions, fmt.Sprintf("AND c.year = $%d", len(args)))

    }

    if price != "" {

        args = append(args, price)

        conditions = append(conditions, fmt.Sprintf("AND c.price <= $%d", len(args)))

    }

    query += strings.Join(conditions, " ")

    rows, err := connection.Query(ctx, query, args...)

    if err != nil {

        return nil, fmt.Errorf("ошибка выполнения запроса: %w", err)

    }

    defer rows.Close()

    var cars []Car

    for rows.Next() {

        var c Car

        err := rows.Scan(&c.ID, &c.BrandName, &c.ModelName, &c.Year, &c.Price)

        if err != nil {

            return nil, fmt.Errorf("ошибка сканирования автомобиля: %w", err)

        }

        cars = append(cars, c)

    }

    if err := rows.Err(); err != nil {

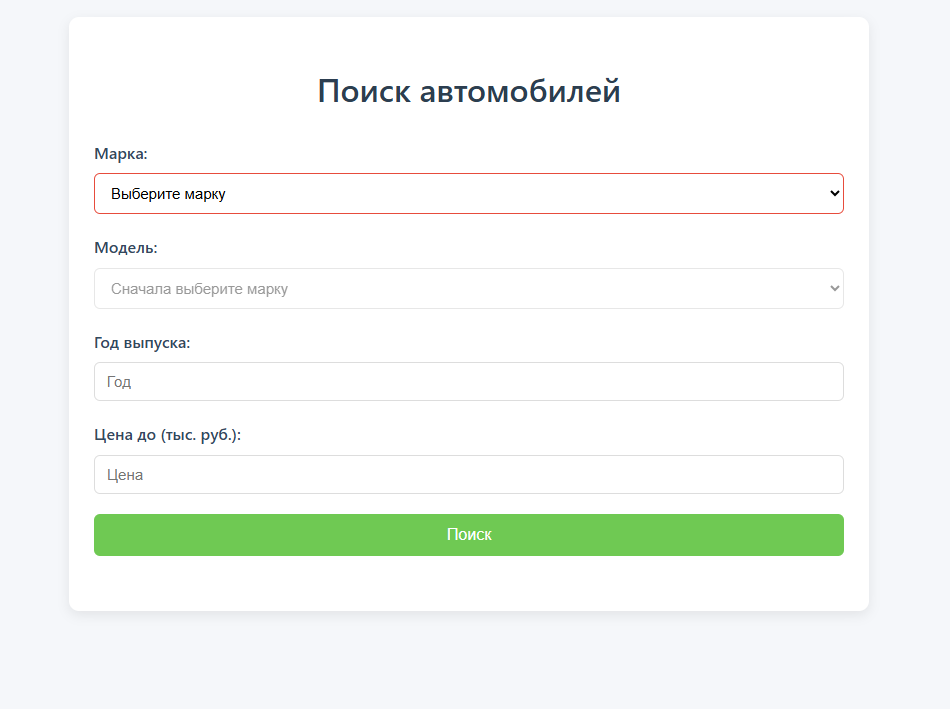
        return nil, fmt.Errorf("ошибка при обработке строк: %w", err)

    }

    return cars, nil

}

# ПРИЛОЖЕНИЕ Б

Скриншот интерфейса:

# ПРИЛОЖЕНИЕ В

GIF-анимация:

