Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский государственный аграрный университет – МСХА имени К. А. Тимирязева»

О.С. Ермолаева, С.В. Пчелинцева

Разработка веб-приложений с интеллектуальными компонентами

Учебно-методическое пособие



Москва
РГАУ-МСХА имени К.А. Тимирязева
2026

УДК 004.8:004.42
ББК 16.333:16.63
Е 74

Рецензенты: ***Егоров И.В.****, к.т.н., доцент кафедры «Техническая механика и мехатроника» СГТУ имени Гагарина и* ***Курашкин С.О.,*** *к.т.н., с.н.с. НОЦ «ФНС России и МГТУ им. Н.Э. Баумана».*

Ермолаева, О.С. Разработка веб-приложений с интеллектуальными компонентами: Учебно-методическое пособие / О.С. Ермолаева, С.В. Пчелинцева; Российский государственный аграрный университет – МСХА имени К. А. Тимирязева. – Москва: РГАУ-МСХА имени К. А. Тимирязева, 2026. - 176 с.

Рекомендовано к изданию учебно-методической комиссией института экономики и управления АПК ФГБОУ ВО РГАУ МСХА имени К.А. Тимирязева, протокол № 2 от 30 сентября 2026 г.

ISBN 978-5-9675-2181-2

Пособие предназначено для студентов направления подготовки 09.03.03 «Прикладная информатика», обучающихся по направленностям «Программные решения для бизнеса» и «Системы искусственного интеллекта» в рамках образовательных программ топ-уровня в сфере искусственного интеллекта. В учебно-методическом пособии рассматриваются вопросы разработки интеллектуальных веб-приложений, охватывающие полный цикл создания: от проектирования пользовательского интерфейса (UI/UX) и статической верстки до интеграции сложных серверных компонентов и модулей искусственного интеллекта (ИИ). Приведены основные определения, архитектура микросервисных систем и принципы работы с данными. Практическое приложение представлено на учебных примерах, основанных на реальных задачах агропромышленного комплекса, что позволяет студентам пройти весь путь от идеи до развертывания работающего продукта.

Издание подготовлено в рамках договора о предоставлении гранта № 70-2025-000753 от 29 мая 2025 г., источником финансового обеспечения которого является субсидия, предоставленная из федерального бюджета, на обеспечение обучения студентов по образовательным программам высшего образования для топ специалистов в сфере искусственного интеллекта (уровня «ДС») по федеральному проекту «Искусственный интеллект» национального проекта «Экономика данных и цифровая трансформация государства».

© Ермолаева О.С., Пчелинцева С.В., 2026
© ФГБОУ ВО РГАУ-МСХА имени К.А. Тимирязева, 2026

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ **5**

ВВЕДЕНИЕ **8**

ГЛАВА 1. ОСНОВЫ ВЕБ ТЕХНОЛОГИЙ И СЕТЕВОГО ВЗАИМОДЕЙСТВИЯ **11**
Архитектура «клиент-сервер» 11
Эталонная модель взаимодействия открытых систем OSI 14
Протоколы HTTP и HTTPS 19
Система доменных имен и структура URL 22
Контрольные вопросы к разделу 25

ГЛАВА 2. ОСНОВЫ ФРОНТЕНД-РАЗРАБОТКИ. HTML5, CSS3, JAVASCRIPT **27**
Основы языка разметки HTML5 27
Теговая модель в HTML 29
Структура тега 29
Структура HTML-документа 30
Каскадные таблицы стилей 32
Основы языка программирования JavaScript 57
Контрольные вопросы к разделу 85

ГЛАВА 3. СЕРВЕРНАЯ РАЗРАБОТКА **87**
Архитектура бэкенда 87
Взаимодействие с фронтендом: REST API 88
Форматы данных для API 89
Базы данных 94
Взаимодействие бэкенда с базой данных 96
Взаимодействие фронтенда с базой данных 98
Коды состояния HTTP 99
Контейнеризация в веб-разработке 100
Контрольные вопросы к разделу 103

ГЛАВА 4. ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ В ВЕБ-РАЗРАБОТКЕ **104**
Применение искусственного интеллекта в веб-приложениях 106
Рекомендательные системы 107
Аналитика поведения пользователей 108
Обработка изображений и текста 109
Автоматизация веб-разработки с помощью ИИ 110
Этические аспекты использования искусственного интеллекта 111
Контрольные вопросы к разделу 113

ГЛАВА 5. ПРАКТИЧЕСКАЯ ЧАСТЬ **114**
Упражнение 1. Макетирование и прототипирование интерфейса. Личный кабинет агронома 114
Контрольные вопросы к упражнению: 117
Упражнение 2 . Статическая верстка HTML 118
Контрольные вопросы к упражнению: 122
Упражнение 3 . Интерактивный интерфейс на JavaScript 123
Контрольные вопросы к упражнению: 127
Упражнение 4. Верстка с CSS и Bootstrap 129
Контрольные вопросы к упражнению: 135
Упражнение 5. Проектирование архитектуры веб-приложения 137
Контрольные вопросы к упражнению: 140
Упражнение 6. Разработка базы данных и SQL-запросы 142

Контрольные вопросы к упражнению: 146
Упражнение 7. Разработка серверной части на FastAPI 147
Контрольные вопросы к упражнению: 153
Упражнение 8. Модуль предиктивной аналитики урожайности 154
Контрольные вопросы к упражнению: 160
Упражнение 9. Интеграция бэкенда и фронтенда 161
Контрольные вопросы к упражнению: 164
Упражнение 10. Контейнеризация приложения с помощью Docker 165
Контрольные вопросы к упражнению: 169
Упражнение 11. Разработка чат-бота 170
Контрольные вопросы к упражнению: 174

БИБЛИОГРАФИЧЕСКИЙ СПИСОК 176

ПРЕДИСЛОВИЕ

Учебное пособие разработано с целью формирования знаний, умений и навыков обучающихся по разработке веб-приложений. Современная цифровая экономика и информационное общество предъявляют принципиально новые требования к специалистам в области информационных технологий. Веб-разработка, некогда бывшая ремеслом по созданию статичных информационных страниц, сегодня трансформировалась в сложнейшую инженерную дисциплину, лежащую в основе функционирования глобальных экосистем, электронной коммерции, социальных коммуникаций и систем государственного управления. Статический контент уступил место динамическим, интерактивным и интеллектуальным веб-приложениям (Single Page Applications, Progressive Web Apps), способным адаптироваться под контекст пользователя и удовлетворять возникающие потребности.

В этом контексте особую значимость приобретает интеграция технологий искусственного интеллекта (ИИ) в стек веб-разработки. Искусственный интеллект перестает быть изолированной областью академических исследований и становится прикладным инструментом, способным радикально повысить эффективность, безопасность и качество пользовательского опыта (User Experience, UX). Персонализация контента, интеллектуальные рекомендательные системы, автоматизация клиентской поддержки через чат-боты, предиктивная аналитика поведения пользователей и обеспечение кибербезопасности — вот лишь неполный перечень задач, решение которых сегодня невозможно без применения методов машинного обучения и анализа данных. Граница между веб-разработчиком и специалистом по данным становится все более проницаемой, формируя разработчиков, способных создавать гибридные системы, объединяющие классические веб-технологии с когнитивными функциями.

Настоящее учебно-методическое пособие разработано с целью формирования у студентов системных знаний и практических компетенций на стыке этих двух динамично развивающихся областей. Дисциплина направлена

на подготовку специалистов, способных не просто владеть инструментарием верстки и программирования, но и проектировать архитектуру интеллектуальных веб-систем, внедрять компоненты ИИ в существующие бизнес-процессы и понимать фундаментальные принципы взаимодействия между фронтендом, бэкендом и моделями машинного обучения.

В результате изучения изложенного в пособии материала студенты должны знать:

- принципы работы клиент-серверной архитектуры и основные интернет-протоколы,
- стек технологий современной веб-разработки: HTML5, CSS3, JavaScript, Python и FastAPI; принципы построения REST API, взаимодействия с базами данных, интеграции клиентской и серверной частей веб-приложения, а также контейнеризации приложений с использованием Docker,
- базовые концепции машинного обучения и области их применения в вебе,
- методы обеспечения информационной безопасности веб-приложений;

уметь:

- разрабатывать адаптивные пользовательские интерфейсы и программировать серверную логику,
- интегрировать в веб-приложение чат-ботов и другие компоненты на базе ИИ,
- использовать внешние программные интерфейсы (API) для расширения функциональности сайта,
- проводить отладку и тестирование разработанных приложений;

владеть:

- навыками, связанными с выбором и применением наиболее эффективных инструментов разработки,
- навыками отладки и тестирования отдельных компонентов программ и приложения в целом.

Пособие может использоваться как для аудиторных практических занятий, так и для самостоятельной работы. Практические задания образуют сквозной проект; при необходимости преподаватель может выбрать отдельные работы или объединить несколько заданий в командный мини-проект.

ВВЕДЕНИЕ

Пособие ориентировано на формирование у студентов практических навыков проектирования и разработки динамических веб-приложений, а также на освоение базовых подходов к интеграции компонентов искусственного интеллекта в клиент-серверную архитектуру. Последовательное изучение теоретического материала и выполнение практических упражнений позволяют сформировать представление о полном цикле создания веб-приложения: от разработки пользовательского интерфейса и настройки серверной логики до работы с источниками данных, внешними программными интерфейсами и интеллектуальной обработкой пользовательских запросов.

Практические задания направлены на закрепление, углубление и расширение знаний о принципах функционирования веб-технологий, протоколах передачи данных HTTP и HTTPS, а также об архитектуре клиент-серверных приложений. В ходе работы обучающиеся осваивают основные элементы современного стека веб-разработки.

Существенной частью курса является знакомство с возможностями применения технологий искусственного интеллекта при решении прикладных задач веб-разработки. Это позволяет сформировать понимание роли AI-компонентов в современных цифровых продуктах, а также освоить начальные навыки их технической интеграции в веб-приложения.

Для выполнения заданий необходим персональный компьютер или рабочая станция с доступом к современному редактору программного кода, например Visual Studio Code, и установленной средой разработки Python версии 3.9 или выше. Допускается использование другого редактора кода, поддерживающего работу с HTML, CSS, JavaScript и Python. Базовым требованием является наличие операционной системы, позволяющей установить необходимое программное обеспечение и запускать разработанные приложения в современном веб-браузере.

Освоение курса предполагает владение базовыми навыками работы в операционной системе на уровне пользователя, понимание основ

алгоритмизации и начальное знание синтаксиса языка программирования Python. Желательным, но не обязательным условием является знакомство с системами контроля версий, в частности с Git. При этом практическая направленность упражнений позволяет обучающимся последовательно развивать недостающие навыки в ходе выполнения заданий.

В процессе работы студентам необходимо создавать структуру проектов, файлы исходного кода, статические ресурсы и иные материалы, используемые в веб-приложениях. Для обеспечения кроссплатформенной совместимости в наименованиях файлов и каталогов рекомендуется использовать только символы латинского алфавита в нижнем регистре, а также разделять составные части имен символом нижнего подчеркивания. Например, файл серверного приложения может иметь имя my_bot.py, а файл каскадных таблиц стилей — путь static/css/style.css.

По итогам выполнения каждого упражнения должен быть подготовлен отчет в текстовом редакторе MS Word. Оформление отчёта рекомендуется выполнять в соответствии с требованиями ГОСТ 7.32–2017, ГОСТ Р 7.0.100–2018 и ГОСТ Р 7.0.5–2008, а также локальными нормативными документами образовательной организации. В отчете необходимо последовательно отразить ход выполнения работы, ключевые этапы создания и настройки приложения, результаты запуска программ и полученные выводы. Описание рекомендуется сопровождать снимками экрана, сделанными в процессе выполнения задания, а при необходимости — фрагментами исходного кода, результатами работы программ, диаграммами архитектуры и иными иллюстративными материалами.

Отдельной частью отчета являются ответы на проверочные вопросы. Они должны быть сформулированы развернуто, содержать ссылку на конкретный этап или пункт задания и демонстрировать понимание примененных технологий. При обосновании ответов допускается использовать фрагменты кода, примеры запросов и ответов веб-сервера, результаты выполнения программ, схемы взаимодействия клиентской и серверной частей приложения, а также иные материалы, подтверждающие ход рассуждения.

Пошаговая организация упражнений обеспечивает последовательное освоение учебного материала и получение планируемых результатов. Вместе с тем обучающимся рекомендуется не ограничиваться строго заданным алгоритмом, а творчески подходить к выполнению практических работ: изменять параметры приложений, экспериментировать с интерфейсом, структурой данных, логикой обработки запросов и возможностями AI-компонентов. Такой подход способствует более глубокому пониманию инструментов веб-разработки, развитию самостоятельности и формированию навыков решения прикладных задач.

ГЛАВА 1. ОСНОВЫ ВЕБ ТЕХНОЛОГИЙ И СЕТЕВОГО ВЗАИМОДЕЙСТВИЯ

Любое современное веб-приложение, от простой визитки до сложной интеллектуальной системы, функционирует в рамках инфраструктуры глобальной сети Интернет. Понятие «Интернет» в контексте веб-разработки выходит далеко за рамки простого определения «глобальной сети». Для разработчика Интернет — это прежде всего программно-аппаратная среда, обладающая набором строгих правил, протоколов и инфраструктурных элементов, которые обеспечивают предсказуемое, надежное и безопасное взаимодействие между распределенными приложениями. Понимание этой среды как единого механизма является ключом к созданию эффективных и отказоустойчивых веб-решений, а понимание фундаментальных принципов этой инфраструктуры является необходимым для веб-разработчика.

Архитектура «клиент-сервер»

В основе функционирования Всемирной паутины и всех современных сетевых приложений лежат два фундаментальных понятия: архитектурная модель взаимодействия «клиент-сервер» и семиуровневая эталонная модель взаимодействия открытых систем (OSI). Понимание этих концепций является обязательным для любого специалиста в области информационных технологий, поскольку они описывают не просто как передаются данные, но и почему сетевые технологии организованы именно таким образом.

Архитектура «клиент-сервер» представляет собой распределенную вычислительную модель, в которой задачи и рабочая нагрузка разделены между двумя основными участниками взаимодействия: поставщиками ресурсов или услуг (серверами) и заказчиками этих ресурсов (клиентами). Эта модель является доминирующей для большинства сетевых приложений, включая веб-технологии, электронную почту, базы данных и файловые службы.

Клиент — это инициатор запроса. В контексте веб-технологий клиентом чаще всего выступает программное обеспечение, установленное на конечном устройстве пользователя, — веб-браузер (например, Яндекс.Браузер, Google Chrome). Основная задача клиента — предоставить пользователю интерфейс для взаимодействия с приложением, сформировать запрос к серверу на основе действий пользователя (например, ввод URL-адреса, нажатие кнопки), отправить этот запрос по сети и, получив ответ, отобразить его в понятном для человека виде. Клиентские устройства, как правило, не обладают значительными вычислительными ресурсами для решения сложных задач и полагаются на серверную часть.

Сервер — это пассивный участник взаимодействия, который находится в режиме постоянного ожидания входящих запросов. Это может быть, как мощный физический компьютер, так и программный комплекс, работающий на нем. В веб-разработке сервер отвечает за хранение данных (файлов сайта, баз данных), обработку бизнес-логики и формирование ответов на запросы клиентов. Получив запрос от клиента, сервер выполняет необходимые действия (например, извлекает информацию из базы данных или выполняет скрипт на языке Python/PHP) и отправляет результат обратно клиенту по тому же каналу связи.

Рассмотрим процесс взаимодействия на примере загрузки веб-страницы.

1. Формирование запроса. Пользователь вводит адрес сайта в браузере. Браузер формирует HTTP-запрос.
2. Передача данных. Запрос инкапсулируется (упаковывается) в соответствии с правилами нижележащих протоколов (TCP, IP) и отправляется по сети на IP-адрес сервера.
3. Обработка на сервере. Серверное ПО принимает запрос, анализирует его и выполняет соответствующую логику.
4. Формирование ответа. Сервер формирует HTTP-ответ, который содержит код состояния (например, 200 OK), заголовки и тело ответа, которым обычно является HTML-код страницы.

5. Передача ответа. Ответ отправляется обратно клиенту по сети.
6. Отображение. Браузер получает данные, интерпретирует HTML, применяет стили CSS и выполняет скрипты JavaScript, отрисовывая страницу на экране.

Архитектура «клиент-сервер» обладает рядом фундаментальных преимуществ, которые и обусловили её доминирование в современных сетевых приложениях. Ключевым достоинством является централизация, при которой все данные и бизнес-логика сосредоточены на сервере, что кардинально упрощает их администрирование, обновление и позволяет обеспечить высокий уровень безопасности за счёт строгого контроля доступа. Кроме того, данная модель обладает высокой масштабируемостью: для обслуживания растущего числа клиентов можно увеличивать мощность сервера или добавлять новые узлы без необходимости модификации клиентского программного обеспечения. Наконец, взаимодействие между клиентом и сервером полностью стандартизировано протоколами, что обеспечивает полную платформенную независимость и позволяет компонентам системы работать на различных операционных системах и аппаратных платформах.

Классическая двухуровневая модель со временем эволюционировала в более сложную трехуровневую архитектуру, которая является стандартом для современных веб-приложений:

Уровень представления (Presentation Tier): Это клиентская часть (Frontend). Отвечает исключительно за пользовательский интерфейс и отображение данных. Технологии: HTML, CSS, JavaScript.

Уровень приложения/логики (Application/Logic Tier): Это серверная часть (Backend). Здесь реализуется вся бизнес-логика приложения, обрабатываются данные, поступившие от клиента, и формируются запросы к базе данных. Технологии: Python (Flask/Django), PHP, Node.js, Java.

Уровень данных (Data Tier): Это системы хранения данных. Отвечает за постоянное и надежное хранение информации. Технологии: системы управления базами данных (СУБД) — PostgreSQL, MySQL, MongoDB.

Такая структура обеспечивает высокую гибкость, например, можно изменить дизайн сайта, не затрагивая логику работы и структуру базы данных.

Эталонная модель взаимодействия открытых систем OSI

Если архитектура «клиент-сервер» описывает кто с кем общается, то модель OSI описывает, как именно происходит это общение на техническом уровне. Модель OSI (Open Systems Interconnection model) — это концептуальная схема, которая стандартизирует функции коммуникационной системы в виде семи абстрактных уровней (рис.1). Она была разработана Международной организацией по стандартизации (ISO) для того, чтобы оборудование и программное обеспечение от разных производителей могли беспрепятственно взаимодействовать друг с другом.

Принцип работы модели основан на строгой иерархии. Каждый уровень модели выполняет свою специфическую задачу и предоставляет сервисы вышележащему уровню, используя сервисы нижележащего уровня. Это называется принципом инкапсуляции: данные с верхнего уровня «заворачиваются» в служебную информацию нижележащего уровня перед отправкой.



Рисунок 1 – Уровни модели OSI

Прикладной уровень (Application Layer) это самый верхний, седьмой уровень модели, который находится в непосредственном взаимодействии с пользовательскими приложениями. Важно понимать, что сам уровень не является конечным приложением (например, браузером или почтовым клиентом), а представляет собой интерфейс, предоставляющий сетевые службы для их работы. Он определяет набор протоколов, с помощью которых приложения получают доступ к ресурсам сети. Ключевыми примерами протоколов данного уровня являются HTTP и HTTPS для передачи гипертекста, FTP для обмена файлами, SMTP для отправки электронной почты и DNS для разрешения доменных имён. Основная функция прикладного уровня заключается в обеспечении доступа пользовательских программ к сетевым службам.

Уровень представления (Presentation Layer) – это шестой уровень, отвечает за синтаксис и семантику передаваемой информации. Его главная задача — обеспечить представление данных в формате, понятном прикладному уровню принимающей системы. Этот уровень выполняет преобразование данных из внутреннего формата приложения в общепринятый сетевой формат и наоборот. Кроме того, на данном уровне осуществляются операции по шифрованию и дешифрованию данных для обеспечения безопасности, а также их сжатие для оптимизации использования канала связи. Примерами функций и стандартов этого уровня служат протоколы SSL/TLS, а также системы кодирования символов, такие как ASCII и UTF-8.

Сеансовый уровень (Session Layer) управляет диалогом между приложениями на различных компьютерах. Он отвечает за установление, поддержание и синхронизацию сеансов связи, а также за их корректное завершение. Этот уровень позволяет добавлять в поток данных контрольные точки (checkpoints), что обеспечивает возможность восстановления передачи с последней точки в случае сбоя. Протоколы сеансового уровня, такие как RPC (удалённый вызов процедур) и NetBIOS, контролируют очередность передачи сообщений и управляют ходом «диалога» между двумя узлами сети.

Транспортный уровень (Transport Layer) обеспечивает прозрачную и надёжную передачу данных между конечными системами. Он отвечает за сегментацию данных, получаемых с верхних уровней, на более мелкие части для передачи по сети. Ключевой функцией этого уровня является контроль доставки: он гарантирует, что все сегменты будут получены в правильной последовательности и без ошибок. Именно здесь определяется тип сервиса: надёжная доставка с установлением соединения (протокол TCP) или быстрая передача без гарантий (протокол UDP). Транспортный уровень также управляет потоком данных, чтобы предотвратить перегрузку принимающего узла.

Сетевой уровень (Network Layer) отвечает за маршрутизацию пакетов данных в составной сети и за управление логической адресацией. Его основная задача — определить оптимальный физический путь для передачи пакета от отправителя к получателю через множество промежуточных сетей и узлов (маршрутизаторов). Для этого используется система логической адресации, наиболее известным примером которой является IP-адрес. Главным протоколом этого уровня является IP (Internet Protocol), который обеспечивает доставку пакетов по указанному адресу.

Канальный уровень (Data Link Layer) обеспечивает передачу данных по физическому каналу связи в пределах одного сегмента сети (например, между двумя соседними узлами). Он упаковывает пакеты, полученные с сетевого уровня, в специальные структуры — кадры (или фреймы). На этом уровне осуществляется физическая адресация устройств с помощью MAC-адресов, а также реализуется механизм доступа к среде передачи для предотвращения коллизий. Канальный уровень также отвечает за обнаружение и, в некоторых случаях, исправление ошибок, возникших на физическом уровне. Примерами технологий этого уровня являются Ethernet, Wi-Fi и PPP.

Физический уровень (Physical Layer) является самым нижним и отвечает за передачу необработанных битовых потоков (последовательностей нулей и единиц) через физическую среду передачи данных. Он определяет все аппаратные и электрические характеристики интерфейсов: уровни напряжений,

скорости передачи данных, типы кабелей и коннекторов (например, медная витая пара или оптоволокно), а также характеристики беспроводной среды (радиоволны). По сути, этот уровень преобразует биты кадра в физические сигналы (электрические, оптические или радиосигналы) и передаёт их по среде от одного узла к другому.

Чтобы понять работу модели OSI на практике, рассмотрим путь данных от веб-браузера к серверу:

Инкапсуляция (на стороне отправителя). Данные создаются на прикладном уровне (например, HTTP-запрос). Спускаясь вниз по стеку OSI:

На 7-м уровне они получают заголовок прикладного протокола.

На 6-м уровне могут быть сжаты или зашифрованы.

На 5-м уровне создается сеанс.

На 4-м уровне (TCP) данные разбиваются на сегменты, каждому из которых присваивается номер порта.

На 3-м уровне (IP) сегменты упаковываются в пакеты (дейтаграммы), к ним добавляется IP-адрес отправителя и получателя.

На 2-м уровне пакеты упаковываются в кадры (фреймы), добавляется MAC-адрес сетевой карты отправителя и следующего узла (например, роутера).

На 1-м уровне кадры преобразуются в последовательность битов (электрических сигналов, световых импульсов) и передаются в кабель или радиоэфир.

Деинкапсуляция (на стороне получателя). Процесс происходит в обратном порядке:

На 1-м уровне биты принимаются и преобразуются обратно в кадры.

На 2-м уровне проверяется MAC-адрес назначения. Если он совпадает с адресом сетевой карты сервера, кадр "разворачивается", извлекается пакет.

На 3-м уровне проверяется IP-адрес назначения. Если он совпадает с IP-адресом сервера, пакет "разворачивается".

Данные поднимаются вверх по стеку до тех пор, пока не достигнут прикладного уровня сервера, где веб-сервер обрабатывает HTTP-запрос.

Архитектура «клиент-сервер» и модель OSI являются двумя сторонами одной медали — сетевого взаимодействия. Первая описывает логическую структуру участников обмена информацией в рамках конкретного приложения (веба), а вторая — универсальный технический стандарт того, как эта информация должна упаковываться и передаваться через сложную иерархию физических и логических сетей.

Протоколы HTTP и HTTPS

В основе функционирования Всемирной паутины (World Wide Web) лежит стек протоколов HTTP/HTTPS. HTTP (HyperText Transfer Protocol) это протокол прикладного уровня, работающий по принципу «запрос-ответ» (request-response). Он является протоколом без сохранения состояния (stateless), что означает: сервер не хранит информацию о предыдущих запросах клиента. Каждый новый запрос обрабатывается как независимая транзакция.

Процесс обмена данными по протоколу HTTP состоит из двух основных этапов. Первый этап — это запрос клиента.

Браузер формирует и отправляет на сервер сообщение-запрос. Он содержит:

- Метод (Method/Verb): Указывает, какое действие необходимо выполнить. Наиболее распространенные методы:

GET: Запросить данные с сервера (например, загрузить веб-страницу). Это самый частый метод.

POST: Отправить данные на сервер для обработки (например, отправить форму с логином и паролем).

PUT: Обновить данные на сервере.

DELETE: Удалить данные на сервере.

- Путь (Path): Указывает на конкретный ресурс на сервере (например, /index.html).
- Версия протокола: Например, HTTP/1.1 или HTTP/2.
- Заголовки (Headers): Служебная информация (метаданные), такая как тип браузера, предпочитаемый язык, типы данных, которые клиент может принять.
- Тело запроса (Body): Содержит данные, отправляемые на сервер (используется в основном в методах POST и PUT).

Сервер обрабатывает запрос и отправляет ответное сообщение (HTTP Response), которое включает (таблица 1):

- Код состояния (Status Code), трехзначное число, информирующее о результате запроса. Коды делятся на группы:

  2xx (Успех): 200 OK — запрос выполнен успешно.

  3xx (Перенаправление): 301 Moved Permanently — ресурс перемещен навсегда.

  4xx (Ошибка клиента): 404 Not Found — запрашиваемый ресурс не найден; 403 Forbidden — доступ запрещен.

  5xx (Ошибка сервера): 500 Internal Server Error — внутренняя ошибка сервера.

- Заголовки ответа. Служебная информация от сервера (например, тип контента Content-Type: text/html).
- Тело ответа. Содержит сам запрошенный ресурс (HTML-код страницы, изображение в формате JPEG и т.д.).

Таблица 1. Пример HTTP-запроса от клиента и HTTP-ответа от сервера

HTTP-запрос от клиента (браузера)	**HTTP-ответ от сервера**
GET /index.html HTTP/1.1 Host: example.com User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.0.0 Safari/537.36 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8 Accept-Language: ru,en;q=0.9 Accept-Encoding: gzip, deflate, br Connection: keep-alive	HTTP/1.1 200 OK Date: Tue, 21 Apr 2026 12:00:00 GMT Server: Apache/2.4.41 (Ubuntu) Content-Type: text/html; charset=UTF-8 Content-Length: 3500 Last-Modified: Mon, 20 Apr 2026 10:30:00 GMT Connection: keep-alive <!DOCTYPE html> <html lang="ru"> <head> <meta charset="UTF-8"> <title>Добро пожаловать</title> </head> <body> Привет, мир! Это содержимое файла index.html. </body> </html>

Несмотря на свою эффективность, у классического HTTP есть критический недостаток: все данные, включая заголовки и тело запроса/ответа, передаются в открытом виде.

HTTPS (HyperText Transfer Protocol Secure) — это не отдельный протокол, а расширение HTTP, которое использует шифрование для защиты передаваемых данных. Буква S в конце означает Secure (безопасный). В современных условиях использование HTTPS является не рекомендацией, а обязательным стандартом для любого веб-сайта. Безопасность в HTTPS обеспечивается за счет использования криптографического протокола TLS, который пришел на смену более старому протоколу SSL.

Ключевым этапом является аутентификация сервера: сервер отправляет клиенту свой цифровой SSL/TLS-сертификат, который содержит его публичный ключ и информацию о владельце домена. Браузер проверяет подлинность этого сертификата, убеждаясь, что он подписан доверенным Центром сертификации (Certificate Authority, CA). Эта проверка гарантирует, что пользователь подключается именно к тому сайту, адрес которого он ввёл, а не к мошенническому двойнику.

Далее, с помощью методов асимметричного шифрования, клиент и сервер безопасно обмениваются данными для создания общего сеансового ключа (симметричного). Именно этот ключ будет использоваться для шифрования всего последующего трафика. На этапе защищённой передачи все данные, включая HTTP-запросы и ответы, шифруются с помощью сеансового ключа перед отправкой и расшифровываются только на принимающей стороне. Для любого перехватчика эти данные будут выглядеть как бессмысленный набор символов.

Использование протокола HTTPS необходимо для обеспечения трёх ключевых аспектов безопасности. Во-первых, это конфиденциальность: шифрование делает невозможным просмотр передаваемых данных (паролей, номеров карт, личных сообщений) третьими лицами, такими как провайдеры или злоумышленники в публичных Wi-Fi сетях. Во-вторых, HTTPS обеспечивает целостность данных, гарантируя, что информация не была изменена или повреждена во время передачи: любая модификация зашифрованных данных сделает их нечитаемыми при расшифровке. Наконец, это аутентификация, при

которой цифровые сертификаты подтверждают подлинность веб-сайта, эффективно защищая пользователей от фишинговых атак.

Система доменных имен и структура URL

Для взаимодействия в сети Интернет каждому устройству и ресурсу необходим уникальный идентификатор, а для пользователя — понятный способ обращения к этим ресурсам. Эту двойную задачу решают три взаимосвязанные концепции: IP-адреса, система доменных имен (DNS) и унифицированные указатели ресурсов (URL).

IP-адрес — это уникальный числовой идентификатор, присваиваемый каждому устройству (компьютеру, серверу, роутеру, смартфону), подключенному к компьютерной сети, которая использует протокол IP для маршрутизации данных. Можно провести аналогию с почтовым адресом, который необходим для точной доставки письма. IP-адрес является конечной целью любого сетевого запроса, так как именно по нему маршрутизаторы в интернете находят путь к целевому серверу.

Существует два основных стандарта IP-адресации: IPv4 (Internet Protocol version 4) и IPv6 (Internet Protocol version 6).

IPv4 является наиболее распространенным стандартом. Адрес представляет собой 32-битное число, обычно записываемое в виде четырех десятичных чисел (октетов), разделенных точками, например, 192.168.1.1.

IPv6 это стандарт, разработанный для решения проблемы нехватки адресов, он использует 128-битную адресацию, что обеспечивает практически неисчерпаемое количество комбинаций. Адрес записывается в виде восьми групп шестнадцатеричных чисел, например, 2001:0db8:85a3:0000:0000:8a2e:0370:7334.

Поскольку человеку крайне сложно запоминать длинные последовательности цифр, была создана система доменных имен (DNS). Это распределенная иерархическая система, которая выполняет функцию «телефонной книги» интернета, преобразуя удобные для человека доменные имена (например, Yandex.ru) в соответствующие им IP-адреса (5.255.255.77).

Процесс разрешения доменного имени (DNS lookup) происходит автоматически и незаметно для пользователя:

- Браузер отправляет запрос на настроенный в системе DNS-резолвер (часто предоставляется интернет-провайдером).
- Если резолвер не знает ответа, он обращается к одному из корневых DNS-серверов.
- Корневой сервер направляет резолвер к DNS-серверу зоны верхнего уровня (например, к серверу, отвечающему за все домены в зоне .ru).
- Сервер зоны направляет резолвер к авторитативному DNS-серверу для конкретного домена (например, для yandex.ru).
- Авторитативный сервер возвращает искомый IP-адрес.
- Резолвер передает IP-адрес браузеру, который теперь может установить соединение с сервером.

Получить домен можно у аккредитованных компаний, которые называются регистраторами доменных имен. Это специализированные организации, имеющие право вносить записи о доменах в центральные реестры.

Первым способом регистрации является получение домена у аккредитованного регистратора. Это самый надежный и правильный способ. Вы работаете напрямую с компанией, которая управляет доменами. В России крупнейшими и наиболее известными регистраторами являются:

REG.RU — один из лидеров рынка с широким выбором доменных зон.

RU-CENTER — старейший и крупнейший регистратор в зоне .ru.

Beget — популярный хостинг-провайдер, который также является аккредитованным регистратором.

Второй возможностью является регистрация через реселлеров. Многие хостинг-провайдеры и конструкторы сайтов предлагают регистрацию доменов как дополнительную услугу. Они выступают посредниками (реселлерами) и регистрируют домен для вас через крупного регистратора. Это удобно, так как вы получаете и хостинг, и домен в одном месте, но иногда цены у реселлеров могут быть выше.

Домены делятся на уровни в зависимости от их положения в иерархии. Для пользователя наиболее важны домены второго уровня — те, что вы непосредственно регистрируете (например, Yandex.ru).

Они, в свою очередь, классифицируются по типу зоны.

Национальные домены — это двухбуквенные домены, закрепленные за конкретными странами.

.ru — Россия.

.рф — Российская Федерация (кириллический домен).

.de — Германия.

.cn — Китай.

.us — США.

Общие домены, которые изначально предназначались для определенных типов организаций, но сейчас доступны всем.

.com (commercial) — самый популярный в мире, изначально для коммерческих организаций.

.org (organization) — для некоммерческих организаций.

.net (network) — для сайтов, связанных с сетевыми технологиями.

.info (information) — для информационных ресурсов.

Тематические или новые домены появились в последние годы и позволяют точнее указать тематику сайта.

Профессиональные: .lawyer, .doctor, .engineer.

Тематические: .shop, .club, .blog, .app, .tech, .design.

Географические: .moscow, .london, .tokyo.

При выборе домена стоит ориентироваться на вашу аудиторию и цели. Для бизнеса в России логично выбирать .ru или .рф. Для международного проекта лучше подойдет .com или тематическая зона, например .global или .online.

URL — это унифицированный указатель ресурса, который представляет собой полный и стандартизированный адрес ресурса в сети. Если DNS отвечает на вопрос «*где* находится ресурс?», то URL отвечает на вопросы «*что* это за ресурс и *как* к нему обратиться?».

Структура URL состоит из нескольких компонентов: <схема>://<авторизация>@<хост>:<порт>/<путь>?<параметры>#<якорь>.

Схема (https://): Указывает протокол, который будет использоваться для доступа к ресурсу.

http:// — стандартный протокол передачи гипертекста.

https:// — защищенная версия HTTP с шифрованием TLS/SSL.

ftp:// — протокол передачи файлов.

mailto: — для создания ссылки на электронную почту.

Авторизация (user:password@) это необязательная часть для передачи логина и пароля. В современном вебе используется редко из соображений безопасности.

Хост (docs.example.com): Доменное имя сервера. Именно эту часть браузер передает в DNS для получения IP-адреса.

Порт (:8080): Указывает конкретный «вход» на сервере. Если порт не указан, используются стандартные порты (80 для HTTP, 443 для HTTPS).

Путь (/articles/ai): Указывает точное местоположение файла или ресурса на сервере.

Параметры запроса (?topic=ml): Набор пар «ключ-значение» для передачи данных на сервер (например, из формы поиска). Начинается с ?, пары разделяются &.

Якорь (#summary): Указывает на конкретное место внутри документа (закладку). Эта часть не отправляется на сервер и используется только браузером для прокрутки страницы.

Контрольные вопросы к разделу

1. Что представляет собой архитектура «клиент-сервер» и каковы её основные компоненты?
2. В чём заключается роль клиента и сервера в веб-взаимодействии?
3. Какие преимущества и недостатки имеет архитектура «клиент-сервер»?
4. Что такое эталонная модель OSI и какие уровни она включает?

5. Какова роль каждого уровня модели OSI в процессе передачи данных?

6. Чем отличается протокол HTTP от HTTPS?

7. Почему использование HTTPS считается более безопасным по сравнению с HTTP?

8. Что такое система доменных имён и какую функцию она выполняет в интернете?

9. Какова структура URL и какие основные его компоненты можно выделить?

10. Как взаимодействуют между собой DNS, HTTP/HTTPS и URL в процессе загрузки веб-страницы?

ГЛАВА 2. ОСНОВЫ ФРОНТЕНД-РАЗРАБОТКИ. HTML5, CSS3, JAVASCRIPT

В современной архитектуре веб-приложений принято чётко разделять обязанности между клиентской и серверной частями. Если серверная часть отвечает за хранение данных, бизнес-логику и обработку запросов, то клиентская часть, или фронтенд, является лицом приложения — именно с ней непосредственно взаимодействует конечный пользователь. Качество этого взаимодействия, скорость отклика интерфейса и общее впечатление от продукта напрямую зависят от того, насколько грамотно реализован фронтенд. Данный раздел пособия посвящён изучению технологического стека, составляющего фундамент веб-страницы или приложения.

Технологическая триада HTML5, CSS3 и JavaScript представляет собой неразрывный комплекс, где каждый компонент выполняет свою уникальную, но взаимодополняющую функцию в создании современного веб-интерфейса. Язык разметки HTML5 служит для описания семантической структуры и содержания документа, формируя логический скелет страницы, на который язык стилей CSS3 накладывает визуальное оформление, отвечая за внешний вид элементов, их расположение и адаптивность. Однако для придания странице интерактивности и динамического поведения необходим язык программирования JavaScript, который позволяет реагировать на действия пользователя и изменять содержимое без перезагрузки, объединяя структуру и стиль в единое, отзывчивое приложение.

Основы языка разметки HTML5

HTML (HyperText Markup Language), или язык гипертекстовой разметки, является фундаментальной технологией Всемирной паутины и используется для создания веб-страниц. Это не язык программирования в классическом понимании, так как он не содержит логики, циклов или переменных, а язык разметки, задача которого — описывать структуру и семантическое содержание веб-документа.

Можно представить HTML-код как скелет или каркас веб-страницы. Он говорит браузеру, что находится на странице: где заголовок, где абзац текста, где изображение, а где — навигационное меню. Ответственность за то, как это выглядит (цвета, шрифты, отступы), лежит на технологии CSS, а за интерактивность и поведение элементов отвечает JavaScript.

Современным стандартом является HTML5, который принес с собой множество новых возможностей: встроенную поддержку аудио и видео, новые семантические элементы для лучшей структуры документа, а также мощные API для создания сложных веб-приложений (например, Canvas для рисования 2D-графики или Web Storage для хранения данных на стороне клиента).

Одной из ключевых особенностей современного HTML является семантическая разметка в стандарте HTML5. Вместо универсальных блоков, таких как , язык предлагает теги, которые чётко описывают своё назначение и смысловую нагрузку. Использование элементов вроде <nav> для навигации или <footer> для «подвала» сайта делает код значительно более читаемым для разработчиков. В то же время это помогает поисковым системам лучше индексировать содержание страницы, а вспомогательным технологиям, например, программам чтения с экрана, — корректно интерпретировать её для людей с ограниченными возможностями.

Другой фундаментальной характеристикой HTML является его иерархическая и вложенная структура. Документ представляет собой древовидную модель, где элементы могут содержать другие элементы, формируя логическую иерархию. Классическим примером служит список <ul>, содержащий пункты <li>. Именно эта вложенность является основой для каскадных таблиц стилей CSS, позволяя применять стили к дочерним элементам, и для языка программирования JavaScript, который использует эту структуру для динамической манипуляции содержимым страницы через объектную модель документа (DOM).

Теговая модель в HTML

Теговая модель — это фундаментальный принцип, лежащий в основе языка HTML. Она определяет, как структурируется и описывается документ. Суть модели заключается в том, что весь контент веб-страницы размечается с помощью специальных меток, называемых тегами. Эти теги указывают браузеру, как интерпретировать и отображать заключенный в них контент. Теги являются основными строительными блоками HTML-документа и формируют его иерархическую структуру.

Структура тега

В HTML существует два основных типа тегов по их структуре:

1. Парные (контейнеры) - это теги, которые имеют как открывающую, так и закрывающую часть. Они предназначены для оборачивания контента и придания ему определенного смысла или стиля. Такие теги, как <html>, <body>, <h1>, <p>, и , используются для описания как блочных элементов (заголовки, параграфы, разделы), так и строчных (фрагменты текста внутри строки).
2. Одиночные (или самозакрывающиеся) теги — это элементы, которые не имеют закрывающей части и не могут содержать вложенный контент, выполняя самостоятельную функцию, например,
 для переноса строки или <img> для вставки изображения. В современном стандарте HTML5 для таких тегов принято использовать синтаксис с прямым слэшем перед закрывающей угловой скобкой (например,
 или), что делает код более строгим, хотя для обеспечения совместимости со старыми браузерами допустим и более короткий вариант без слэша.

Теги могут вкладываться друг в друга, создавая древовидную иерархическую структуру. Это называется вложенностью. При этом необходимо соблюдать главное правило, что теги должны закрываться в

порядке, обратном их открытию. То есть, первый открытый тег должен быть закрыт последним. Пример: Это важный текст.

Таким образом, теговая модель превращает плоский текст в структурированный документ, где каждый элемент имеет свое место и значение, что позволяет браузеру корректно отобразить страницу, а другим технологиям (CSS, JavaScript) — эффективно взаимодействовать с её элементами.

Структура HTML-документа

Любая веб-страница, независимо от её сложности, начинается с фундаментальной и строго определённой структуры. Понимание этой структуры является первым и самым важным шагом в изучении HTML, поскольку она задает правила, по которым браузер интерпретирует и отображает содержимое. Стандартный HTML-документ представляет собой иерархию вложенных элементов, каждый из которых выполняет свою уникальную роль.

1. Декларация типа документа. Самый первый элемент в любом HTML-файле — это декларация <!DOCTYPE>. Она не является HTML-тегом, а представляет собой инструкцию для браузера, сообщающую ему, какая версия HTML используется в документе. Это гарантирует, что страница будет отрисована в соответствии с современными спецификациями HTML и CSS. В HTML5 декларация была значительно упрощена и выглядит так: <!DOCTYPE html>. Эту строку легко запомнить, и её необходимо размещать на самой первой строке документа, перед открывающим тегом <html>.

2. Корневой элемент (<html>). Сразу после декларации следует тег <html>, который является корневым элементом всего документа. Все остальные элементы должны находиться внутри него. Он сообщает браузеру, что всё его содержимое является HTML-кодом. Внутри тега присутствует атрибут lang, который указывает язык документа (например, lang="ru" для русского).

Наличие его помогает поисковым системам, программам для чтения с экрана и другим инструментам правильно обрабатывать контент.

3. Элемент заголовок (<head>) содержит метаинформацию о документе. Содержимое этого блока не отображается непосредственно на веб-странице, но предоставляет браузеру и поисковым системам важные инструкции. В <head> указываются следующие элементы:

<title> Задает заголовок страницы, который отображается на вкладке браузера и используется в качестве заголовка при сохранении страницы в закладки.

<meta charset="UTF-8"> Определяет кодировку документа. UTF-8 является стандартом де-факто, так как поддерживает символы практически всех языков мира, включая кириллицу.

<link> Используется для подключения внешних ресурсов, чаще всего — файлов каскадных таблиц стилей, дополнительных библиотек.

4. Тело документа (<body>) содержит всё видимое содержимое веб-страницы, которое будет отображено в окне браузера: текст, изображения, видео, ссылки и другие элементы. Именно внутри <body> располагается вся структура и контент, которые видит пользователь, а также элемент <script> , который позволяет подключать или встраивать код на языке JavaScript.

Собрав все элементы вместе, мы получаем шаблон для любой HTML-страницы:

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="UTF-8">
    <title>Моя первая веб-страница</title>
    <!-- Здесь подключаются CSS и мета-теги -->
  </head>
  <body>
    <h1>Привет, мир!</h1>
    <p>Это содержимое страницы.</p>
```

```
    <!-- Здесь находится весь видимый контент -->
  </body>
</html>
```

Каскадные таблицы стилей

Если язык разметки HTML отвечает за создание «скелета» веб-страницы, определяя её семантическую структуру и содержание, то CSS (Cascading Style Sheets — каскадные таблицы стилей) является её «одеждой». Это технология, превращает иерархию текстовых элементов в визуально привлекательный и интуитивно понятный пользовательский интерфейс. Без CSS все веб-сайты выглядели бы как простые текстовые документы, лишенные индивидуальности, фирменного стиля и удобства использования.

Основная задача CSS — разделение содержания и представления. Такое разделение является фундаментальным принципом современной веб-разработки. Оно позволяет изменять дизайн всего сайта, редактируя один-единственный CSS-файл, не затрагивая при этом HTML-разметку. Это кардинально упрощает поддержку, масштабирование и редизайн проектов.

В этом разделе мы познакомимся с синтаксисом языка CSS и основными способами его подключения к HTML-документу. Мы изучим, как с помощью селекторов точно нацеливаться на нужные элементы страницы и изменять их внешний вид. Особое внимание будет уделено каскадированию и наследованию — двум ключевым механизмам, которые определяют, какое именно стилевое правило будет применено к элементу в случае конфликта. Мы начнём с основ: управления цветами, шрифтами, отступами и границами, чтобы заложить прочный фундамент для дальнейшего изучения более сложных концепций верстки.

Синтаксис CSS и способы подключения

Стилевое правило в CSS состоит из селектора и блока объявлений.

Селектор указывает, к какому HTML-элементу (или группе элементов) применить стили.

Блок объявлений заключается в фигурные скобки {...} и содержит одно или несколько объявлений.

Объявление — это пара «свойство: значение;», где свойство (например, color или font-size) определяет, какой именно аспект стиля мы меняем, а значение (например, red или 16px) — как именно мы его меняем.

```
p {
   color: blue; /* Свойство color для всех параграфов станет blue */
   font-size: 16px; /* Свойство font-size станет 16 пикселей */
}
```

Существует три основных способа подключения CSS-стилей к HTML-документу:

1. Внешние таблицы стилей. Стили выносятся в отдельный файл с расширением .css, который затем подключается ко всем страницам сайта. Это обеспечивает централизованное управление дизайном.

```
<!-- Внутри тега <head> -->
<link rel="stylesheet" href="styles.css">
```

2. Внутренние таблицы стилей. Стили определяются внутри тега <style> в секции <head> документа. Этот способ подходит для стилей, уникальных для одной страницы.

```
<head>
   <style>
      h1 {
         color: green;
      }
   </style>
</head>
```

3. Инлайновые (встроенные) стили. Стили применяются непосредственно к конкретному элементу через атрибут style. Этот метод имеет наивысший приоритет, но его использование для основной верстки считается плохой практикой, так как он смешивает структуру и представление.

```
<p style="color: red; font-size: 20px;">Этот текст будет красным и крупным.</p>
```

Каскадирование в CSS

Одной из ключевых особенностей CSS, отраженной в его названии, является механизм каскадирования. Когда к одному и тому же HTML-элементу применяется несколько противоречащих друг другу правил, браузер должен решить, какое из них имеет приоритет. Этот процесс определяется набором четких правил, которые учитывают источник стиля, специфичность селектора и порядок объявления.

Рассмотрим эти правила в порядке их значимости.

1. Источник стиля (Origin) является самым высоким уровнем приоритета. Браузер определяет, откуда пришло правило, и выстраивает иерархию:

- Стили браузера (User Agent Styles). Это стили по умолчанию, которые встроены в сам браузер. Они применяются, если для элемента не задано никаких других правил. Например, по умолчанию ссылки (<a>) синие и подчеркнутые.
- Стили автора (Author Styles). Это стили, которые пишет разработчик сайта. К ним относятся правила из внешних файлов (.css), внутренних блоков <style> и инлайновых атрибутов style. Это самый распространенный тип стилей.
- Стили пользователя (User Styles). Это стили, которые настраивает сам пользователь в своем браузере (например, для повышения доступности или по личным предпочтениям).
- Важные стили (!important). Любое правило, помеченное как !important, получает наивысший приоритет, перекрывая даже стили пользователя (за редкими исключениями для специальных возможностей). Использование `!important` считается плохой практикой, так как оно ломает естественный каскад и усложняет отладку кода, поэтому его следует избегать.

2. Специфичность селектора (Selector Specificity). Если конфликтующие правила имеют одинаковый источник и не используют !important, браузер сравнивает их **специфичность**. Специфичность — это вес или "сила"

селектора. Она рассчитывается в виде четырехзначного числа (a, b, c, d). Побеждает то правило, у которого специфичность выше.

a (1000): Встроенные стили (<p style="...">). Имеют наивысшую специфичность.

b (0-100): Количество ID-селекторов (#header).

c (0-10): Количество классов, псевдоклассов и атрибутов (.menu, :hover, [type="text"]).

d (0-10): Количество тегов и псевдоэлементов (div, h1, ::before).

3. Порядок объявления (Source Order). Если и источник стиля, и специфичность селекторов у конфликтующих правил одинаковы, в силу вступает последнее правило в коде. То, что написано ниже, переопределяет то, что написано выше.

```
/* В файле styles.css */
h1 { color: blue; } /* Написано первым */
h1 { color: green; } /* Написано вторым */
```

В результате выполнения кода примера заголовок <h1> будет зеленым.

Когда браузер применяет стили к элементу, он следует итоговому алгоритму:

- найти все правила, которые применяются к данному элементу;
- отбросить все правила с !important, кроме тех, что имеют наивысший приоритет по источнику;
- среди оставшихся правил выбрать те, у которых наибольшая специфичность селектора;
- если несколько правил имеют одинаковую специфичность, выбрать то, которое было объявлено позже в коде;
- если все еще есть конфликт (например, два одинаковых селектора в одном файле), применить последнее;
- если ни одно правило не подошло, применить стиль браузера по умолчанию.

Наследование в CSS

Наследование — это один из фундаментальных механизмов в CSS, который определяет, как некоторые свойства передаются от родительских элементов к их потомкам (дочерним, внучатым и т.д. элементам). Понимание этого принципа позволяет писать более чистый, лаконичный и эффективный код, избегая ненужного дублирования стилей.

Представьте HTML-документ как дерево, где <html> — это корень, а все остальные элементы — его ветви. Когда вы применяете стиль к родительскому элементу, некоторые из его свойств автоматически применяются вниз по дереву ко всем вложенным в него элементам.

```
<body style="color: blue; font-family: Arial, sans-serif;">
  <h1>Заголовок</h1>
  <p>Это абзац текста. <span>Этот span тоже будет синим.</span></p>
</body>
```

В этом примере текст внутри тегов <h1>, <p> и будет синим и отобразится шрифтом Arial. Мы не прописывали эти стили для каждого элемента в отдельности, но они унаследовали их от своего родителя — <body>.

Не все CSS-свойства наследуются. Это сделано для логического разделения. Например, дочернему абзацу не нужно наследовать от родителя border (границу) или margin (внешний отступ). Как правило, свойства, отвечающие за блочную модель (размеры, отступы, границы) и позиционирование не наследуются.

CSS предоставляет возможности управления процессом наследованием разработчику в виде двух специальных значений для свойства inherit.

- Inherit можно применить к любому свойству любого элемента. Оно заставляет элемент принудительно унаследовать значение этого свойства от его родителя, даже если по умолчанию это свойство не наследуется.

Пример: По умолчанию у элемента <body> нет фона, и фон его дочернего будет прозрачным. Чтобы заставить унаследовать фон от <body>, нужно написать:

```
body {
   background-color: lightgray;
}
.content {
   background-color: inherit; /* Теперь у .content будет фон lightgray */
}
```

- Initial сбрасывает свойство к его начальному (дефолтному) значению, как определено в спецификации CSS. Это не то же самое, что отсутствие стиля.
- Revert значение сбрасывает свойство до того значения, которое оно имело бы в таблице стилей браузера. Полезно для отмены стилей, заданных автором.

Модель визуального форматирования. Блочная модель

Фундаментальной концепцией в CSS, необходимой для понимания верстки и управления пространством на веб-странице, является модель визуального форматирования, а ее ключевой составляющей — блочная модель.

В CSS каждый элемент на странице, будь то абзац, изображение или , рассматривается как прямоугольный блок. Блочная модель описывает, из каких частей состоит этот прямоугольник и как они влияют на общие размеры и положение элемента в макете (рис.2).

Понимание блочной модели критически важно, так как именно она объясняет, почему элементы могут иметь размеры, отличные от заданных через свойства width и height, и почему они могут "наезжать" друг на друга.

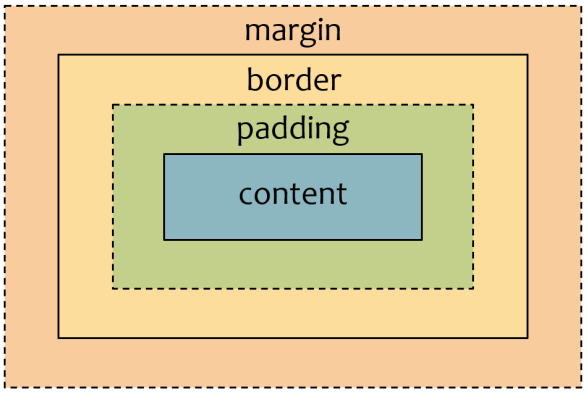


Рисунок 2. Структура блочной модели

Блочная модель состоит из четырех слоев, которые идут от центра к краю.

Контент (content) — это область блока, где находится его содержимое: текст, изображение или другие вложенные элементы. Размеры этой области можно задавать с помощью свойств width (ширина) и height (высота).

Внутренний отступ (padding) это пространство между областью контента и границей элемента. В отличие от внешних отступов, внутренний отступ является частью самого элемента. Он имеет тот же фон (цвет, изображение), что и элемент. Свойства для управления:

padding-top, padding-right, padding-bottom, padding-left

Краткая запись: padding: 10px 20px 15px 5px; (по часовой стрелке: верх, право, низ, лево).

Граница (border) — это линия, которая окружает элемент вместе с его внутренним отступом. Граница может иметь разную толщину (border-width), стиль (border-style: сплошная, пунктирная и т.д.) и цвет (border-color).

Внешний отступ (margin) это пространство за пределами границы элемента. Внешний отступ используется для отделения одного блока от другого. Он является прозрачным и не наследует фоновый цвет элемента.

Свойства: margin-top, margin-right, margin-bottom, margin-left.

Важная особенность: вертикальные внешние отступы (margin-top и margin-bottom) между соседними блочными элементами могут "схлопываться" (collapse), образуя один общий отступ, равный большему из них.

Общий размер, который элемент занимает на странице, складывается из всех этих частей.

Общая ширина = left-margin + left-border + left-padding + width + right-padding + right-border + right-margin.

Общая высота = top-margin + top-border + top-padding + height + bottom-padding + bottom-border + bottom-margin.

Свойство box-sizing

Исторически в CSS (стандарт CSS1) свойство width означало ширину только области контента. Это часто приводило к путанице: если вы устанавливали блоку width: 100px; padding: 10px;, его реальная ширина становилась 120 пикселей (100 + 10 слева + 10 справа).

В CSS3 было введено свойство box-sizing, которое позволяет изменить это поведение.

box-sizing: content-box; (значение по умолчанию) Ширина и высота (width, height) применяются только к области контента. Отступы (padding) и границы (border) добавляются к этой ширине, увеличивая общий размер блока.

box-sizing: border-box; Ширина и высота (width, height) включают в себя и область контента, и внутренние отступы (padding), и границы (border). Это поведение гораздо интуитивнее для верстки сеток и адаптивных макетов.

Пример:

```
/* Для обоих блоков заданы одинаковые параметры */
.box {
   width: 200px;
   padding: 20px;
   border: 5px solid black;
}

/* Вариант 1: Поведение по умолчанию */
.box-default {
   box-sizing: content-box; /* Необязательно, так как это значение по умолчанию */
   /* Реальная ширина = 200px (width) + 40px (padding) + 10px (border) = 250px */
}

/* Вариант 2: Интуитивное поведение */
.box-intuitive {
   box-sizing: border-box;
   /* Реальная ширина = 200px. Из них вычитаются padding и border */
}
```

В современной верстке считается хорошей практикой применять box-sizing: border-box; ко всем элементам на странице для предсказуемости расчетов.

Позиционирование и верстка

Для управления расположением элементов на странице используются различные типы позиционирования (position) (табл.2) и современные методы верстки.

Таблица 2. Типы позиционирования

Тип	**Позиционирование элементов**
static (по умолчанию)	Элементы располагаются в обычном потоке документа
relative	Элемент смещается относительно своего обычного положения, но место, которое он занимал, сохраняется
absolute	Элемент извлекается из потока документа и позиционируется относительно ближайшего родительского элемента с позиционированием, отличным от static
fixed	Элемент позиционируется относительно окна браузера и остается на месте при прокрутке страницы

Современные методы верстки

Современным и мощным модулем CSS, предназначенным для создания гибких и эффективных макетов является Flexbox. Его главная задача — предоставить более эффективный способ выравнивания, распределения пространства и упорядочивания элементов внутри контейнера, даже когда их размер неизвестен или динамически меняется (что часто встречается в адаптивном дизайне).

Ключевая особенность Flexbox заключается в том, что это одномерная модель верстки. Это означает, что он управляет элементами в одном направлении — либо в строке (горизонтально), либо в столбце (вертикально). Для сложных двумерных сеток (где нужно управлять и строками, и столбцами одновременно) существует другой, более мощный инструмент — CSS Grid Layout.

Чтобы начать использовать Flexbox, необходимо выполнить два простых шага: создать контейнер, для чего осуществить выбор родительского элемента, который будет содержать flex-элементы, и применить свойство display. Для этого элемента нужно задать значение display: flex; или display: inline-flex.

```
/* Создаем flex-контейнер со всеми его дочерними элементами */
.container {
   display: flex; /* или inline-flex; */
}
```

Как только вы примените display: flex; к родительскому элементу, все его прямые дочерние элементы автоматически становятся flex-элементами и элементы выстраиваются в ряд (вдоль главной оси) по умолчанию.

Понимание осей — ключ к освоению Flexbox. В отличие от обычного потока документа, где элементы идут строго сверху вниз, во Flexbox есть две перпендикулярные оси. Все свойства, связанные с выравниванием (justify-content, align-items), работают либо вдоль главной, либо вдоль поперечной оси.

По умолчанию главная ось (main axis) направлена горизонтально слева направо. Направление главной оси можно менять с помощью свойства flex-direction.

Поперечная ось (cross axis) всегда перпендикулярна главной оси. По умолчанию — вертикальная.

Flexbox предоставляет набор свойств для управления расположением элементов вдоль обеих осей.

Таблица 3 — Свойства управления направлением и визуальным порядком flex-элементов

Свойство	**Значение**	**Описание**
flex-direction	row	Значение по умолчанию. Главная ось направлена горизонтально; элементы располагаются слева направо при направлении текста слева направо
flex-direction	row-reverse	Главная ось направлена горизонтально; визуальный порядок элементов меняется на обратный
flex-direction	column	Главная ось направлена вертикально; элементы располагаются сверху вниз
flex-direction	column-reverse	Главная ось направлена вертикально; визуальный порядок элементов меняется на обратный
order	целое число	Определяет порядковую группу flex-элемента. По умолчанию все элементы имеют значение order: 0; сначала отображаются элементы с меньшим значением order

Следует учитывать, что использование свойства order изменяет только визуальное представление элементов. Последовательность элементов в HTML-коде, порядок обхода документа и логика доступности страницы при этом могут не измениться. Поэтому не рекомендуется применять order для

перестановки содержимого, если от порядка элементов зависит смысл текста или последовательность взаимодействия пользователя с интерфейсом.

Свойство justify-content управляет тем, как flex-элементы располагаются вдоль главной оси, если в контейнере остается свободное пространство. Направление действия свойства зависит от значения flex-direction. Например, при flex-direction: row выравнивание происходит по горизонтали, а при flex-direction: column — по вертикали.

В таблице 4 приведены распространенные значения свойства justify-content.

Таблица 4 — Значения свойства justify-content для выравнивания flex-элементов вдоль главной оси

Значение	**Описание**
flex-start	Значение по умолчанию. Элементы располагаются у начала главной оси
flex-end	Элементы располагаются у конца главной оси
center	Элементы группируются в центре главной оси
space-between	Первый элемент располагается у начала оси, последний — у конца; свободное пространство равномерно распределяется между элементами
space-around	Вокруг каждого элемента создается свободное пространство; у внешних границ контейнера отступ получается в два раза меньше промежутка между соседними элементами
space-evenly	Свободное пространство равномерно распределяется между элементами и между крайними элементами и границами контейнера

Свойство align-items задает выравнивание flex-элементов вдоль поперечной оси. При flex-direction: row это обычно вертикальное выравнивание, а при flex-direction: column — горизонтальное.

Основные значения свойства align-items приведены в таблице 5.

Таблица 5 — Значения свойства align-items для выравнивания flex-элементов вдоль поперечной оси

Значение	**Описание**
stretch	Значение по умолчанию. Элементы растягиваются вдоль поперечной оси, если их размер в этом направлении задан как auto
flex-start	Элементы располагаются у начала поперечной оси

flex-end	Элементы располагаются у конца поперечной оси
center	Элементы выравниваются по центру поперечной оси
baseline	Элементы выравниваются по базовой линии текста

Управление гибкостью элементов. Свойство flex является сокращенной записью для свойств flex-grow, flex-shrink и flex-basis. Эти параметры определяют, как flex-элемент ведет себя при наличии свободного места в контейнере или при недостатке доступного пространства.

Основные свойства управления размером flex-элементов приведены в таблице 6.

Таблица 6 — Свойства управления гибкостью размеров flex-элементов

Свойство	**Назначение**	**Пример**
flex-grow	Определяет, какую долю свободного пространства может занять элемент. Значение 0 запрещает элементу увеличиваться; значение 1 разрешает рост	flex-grow: 1;
flex-shrink	Определяет коэффициент сжатия элемента при недостатке места в контейнере. Значение 0 запрещает сжатие	flex-shrink: 1;
flex-basis	Задает исходный размер элемента вдоль главной оси до распределения свободного пространства	flex-basis: 200px;
flex	Сокращенная запись для свойств flex-grow, flex-shrink и flex-basis	flex: 1 1 200px;

Если у двух элементов значения flex-grow равны 1 и 2, то второй элемент получит в два раза большую долю свободного пространства, а не обязательно станет в два раза шире первого. Это различие особенно важно, если элементы имеют разные значения flex-basis или уже занимают разное пространство до его распределения.

Flexbox рекомендуется использовать в тех случаях, когда необходимо расположить элементы в одном измерении: в строку или столбец. Этот механизм особенно удобен для решения следующих задач:

- центрирование элемента по горизонтали и вертикали;
- создание навигационных панелей, меню и панелей инструментов;

- размещение кнопок, значков и текстовых элементов в одной строке;
- выравнивание карточек товаров или информационных блоков в каталоге;
- построение строк форм, содержащих подписи, поля ввода и кнопки;
- создание макета страницы с «липким» нижним колонтитулом, размещаемым в нижней части окна браузера при недостатке содержимого;
- распределение свободного пространства между элементами интерфейса.

Для более сложных двумерных макетов, где необходимо одновременно управлять строками и столбцами, как правило, более подходящим инструментом является CSS Grid. **CSS Grid Layout (Grid)** — это полноценная двумерная система верстки. Она позволяет одновременно контролировать расположение элементов и по горизонтали (строки), и по вертикали (столбцы), что делает ее идеальным выбором для создания сложных макетов целых страниц: шапки, основного контента, боковых колонок и футера.

Как и во Flexbox, все начинается с объявления контейнера. Создается родительский элемент, который будет содержать все элементы вашего макета, затем к нему применяем свойства display: grid; или display: inline-grid;. Как только элемент становится grid-контейнером, все его прямые дочерние элементы автоматически становятся grid-элементами и начинают располагаться в соответствии с правилами сетки.

Grid представляет собой сетку из невидимых линий, которые образуют строки и столбцы. Элементы размещаются в ячейках, образованных на пересечении этих строк и столбцов.

Определение строк и столбцов. Для создания структуры сетки используются два основных свойства: grid-template-columns (определяет количество и ширину столбцов) и grid-template-rows (определяет количество и высоту строк).

В качестве значений ширины и высоты можно использовать любые единицы длины (px, em, %) или, что более гибко, фракционные единицы (fr). Единица 1fr означает "одну часть из доступного свободного пространства".

```
.container {
   display: grid;
   /* Создаем 12 колонок равной ширины */
   grid-template-columns: repeat(12, 1fr);
   /* Задаем высоту первой строки (шапка) */
   grid-template-rows: 80px;
   gap: 20px; /* Расстояние между элементами (новое свойство gap) */
}
```

Существует несколько способов указать, где именно должен находиться элемент.

1. Автоматическое размещение (по умолчанию). Если вы не указываете элементам их точное местоположение, Grid будет размещать их по порядку в доступных ячейках. Это поведение отлично подходит для простых списков или галерей изображений.

2. Явное размещение с помощью grid-column и grid-row. Это контролируемый метод. Свойства grid-column и grid-row позволяют указать, между какими линиями сетки должен располагаться элемент. Синтаксис: grid-column: <линия_начала> / <линия_конца>; Линии нумеруются, начиная с 1.

Пример: Макет страницы с шапкой, контентом и футером. Представим сетку из 12 колонок и 3 строк.

```
<div class="page">
   <header>Шапка</header>
   <aside>Боковая панель</aside>
   <main>Основной контент</main>
   <footer>Подвал</footer>
</div>


.page {
   display: grid;
   grid-template-columns: repeat(12, 1fr);
   grid-template-rows: auto 1fr auto; /* Шапка - авто, контент - все остальное, футер - авто */
   gap: 20px;
}

/* Шапка занимает все 12 колонок в первой строке */
header {
   grid-column: 1 / -1; /* От первой до последней линии (-1 - удобный способ) */
}

/* Боковая панель занимает первые 3 колонки во второй строке */
aside {
```

```
    grid-column: 1 / 4; /* Занимает колонки со 2-й по 3-ю включительно */
}

/* Основной контент занимает оставшиеся 9 колонок (с 4-й по 12-ю) */
main {
    grid-column: 4 / -1;
}

/* Футер, как и шапка, занимает всю ширину */
footer {
    grid-column: 1 / -1;
}
```

CSS Grid Layout — это современный стандарт для создания сложных и адаптивных веб-макетов, который делает процесс верстки более интуитивным и мощным. Его ключевое преимущество — двумерность, обеспечивающая полный контроль над расположением элементов по осям X и Y одновременно, что, в отличие от Flexbox, позволяет легко создавать гибкие и асимметричные дизайны. При этом логика верстки полностью отделена от HTML-разметки, что избавляет от необходимости добавлять лишние теги для создания колонок, а сама сетка может быть легко перестроена с помощью медиа-запросов для адаптации под любые экраны.

Адаптивный дизайн и медиа-запросы

В эпоху многообразия устройств — от небольших смартфонов до огромных 4K-мониторов — создание веб-страниц с фиксированной шириной стало неэффективным. Пользовательский опыт на мобильном телефоне при просмотре сайта, созданного для настольного компьютера, будет крайне негативным: текст будет слишком мелким, элементы навигации — нечитаемыми, а горизонтальная прокрутка — неизбежной.

Для решения этой проблемы были разработаны подходы адаптивного и отзывчивого дизайна, а ключевым инструментом для их реализации в CSS стали медиа-запросы.

Адаптивный веб-дизайн — это подход, при котором дизайн и разработка сайта должны реагировать на поведение пользователя и окружающую среду (платформу, размер экрана, ориентацию устройства).

В основе адаптивного дизайна лежат три главных принципа:

- Гибкая сетка. Вместо пикселей для разметки макета используются относительные единицы, такие как проценты (%), что позволяет контейнеру растягиваться и сжиматься в зависимости от ширины экрана.
- Гибкие изображения и медиа. Изображения и другие медиа-элементы не должны превышать ширину своего родительского контейнера. Это достигается простым правилом: img, video, object {max-width: 100%; height: auto;}.
- Медиа-запросы, механизм, позволяющий применять разные стили для разных условий просмотра.

Медиа-запросы: синтаксис и применение

Медиа-запрос — это логическое условие, которое проверяет определенные параметры устройства или окна браузера. Если условие истинно, то стили, заключенные внутри медиа-запроса, будут применены.

```
/* Если условие истинно, применить эти стили */
@media (условие) {
  /* CSS-правила */
  .container {
    width: 100%;
  }
  h1 {
    font-size: 24px;
  }
}
```

Основные медиа-функции (условия) представлены в таблице 7.

Таблица 7 — Основные медиа-функции CSS для адаптации интерфейса к параметрам устройства

Медиа-функция	**Назначение**
width и height	Определяют ширину и высоту области просмотра браузера (viewport). Наиболее часто используются в медиазапросах для настройки интерфейса под различные размеры экранов
orientation	Определяет ориентацию области просмотра: portrait — портретная ориентация, при которой высота больше или равна ширине; landscape — альбомная ориентация, при которой ширина больше высоты
resolution	Определяет плотность пикселей устройства вывода. Может использоваться при подготовке изображений и графических элементов для экранов с повышенной плотностью пикселей, включая Retina-дисплеи

Современная и рекомендуемая методология придерживается разработки сначала для мобильных устройств (с самым маленьким экраном), а затем постепенное усложнение макета для больших экранов.

1. Сначала создаем стили для мобильных устройств. Эти стили будут работать на всех экранах.
2. Добавляем точки излома (*Breakpoints*). С помощью медиа-запросов min-width мы добавляем новые стили только тогда, когда ширина экрана становится больше определенного значения.

Пример реализации:

```
/* 1. Базовые стили для мобильных устройств */
body {
   font-size: 16px;
}
.container {
   padding: 10px;
   width: 100%; /* Занимает всю ширину на маленьких экранах */
}

/* 2. Стили для планшетов (min-width: 768px) */
@media (min-width: 768px) {
   .container {
      width: 750px; /* Фиксированная ширина для планшетов */
      margin: 0 auto; /* Центрируем контейнер */
   }
}

/* 3. Стили для настольных компьютеров (min-width: 1024px) */
@media (min-width: 1024px) {
   .container {
      width: 960px; /* Еще большая ширина для ПК */
   }
   nav ul {
      display: flex; /* Используем Flexbox для горизонтального меню только на ПК */
      gap: 20px;
   }
}
```

Хотя конкретные значения точек останова зависят от структуры интерфейса, целевой аудитории и содержимого веб-страницы, в адаптивной верстке традиционно используются ориентировочные диапазоны ширины области просмотра. Они представлены в таблице 8.

Таблица 8 — Диапазоны ширины области просмотра

Диапазон ширины области просмотра	Тип устройства или сценарий отображения
От 320 px до 480 px	Мобильные телефоны
От 481 px до 767 px	Крупные смартфоны и компактные планшеты
От 768 px до 1023 px	Планшеты в альбомной ориентации и компактные ноутбуки
От 1024 px до 1199 px	Стандартные ноутбуки и мониторы небольшого размера
1200 px и более	Широкоформатные настольные мониторы

Использование медиа-запросов позволяет создавать единый сайт, который обеспечивает отличный пользовательский опыт на любом устройстве, что является стандартом современной веб-разработки.

Псевдоклассы и псевдоэлементы

В CSS существуют специальные селекторы, которые позволяют стилизовать элементы на основе их состояния или создавать и стилизовать виртуальные элементы, отсутствующие в HTML-разметке. Эти инструменты значительно расширяют возможности разработчика, позволяя создавать интерактивные и детализированные интерфейсы без необходимости добавлять лишние классы и элементы в HTML.

Псевдоклассы — это селекторы, которые применяются к элементам в определенном состоянии. Они позволяют стилизовать элемент в зависимости от действий пользователя, положения в документе или истории навигации.

Псевдокласс всегда добавляется к основному селектору через двоеточие :. Существует несколько основных групп псевдоклассов.

1. Псевдоклассы взаимодействия пользователя, котоые реагируют на действия пользователя с элементом представлены в таблице 9.

Таблица 9 — Псевдоклассы CSS и их применение

Псевдокласс	Назначение	Пример применения
:hover	Применяется при наведении курсора на элемент	Изменение цвета кнопки при наведении

:active	Применяется в момент активации элемента, например при нажатии и удержании кнопки мыши	Создание эффекта нажатия кнопки
:focus	Применяется, когда элемент получает фокус: при клике или переходе с клавиатуры	Выделение активного поля ввода

```
/* Стили для ссылки при наведении и нажатии */
a:hover {
   color: red;
   text-decoration: underline;
}
a:active {
   color: darkred;
   transform: scale(0.98);
}

/* Стили для поля ввода при фокусе */
input:focus {
   border-color: blue;
   outline: none;
}
```

2. Структурные псевдоклассы (таблица 10) позволяют выбирать элементы на основе их положения в дереве HTML.

Таблица 10 — Структурные псевдоклассы CSS

Псевдокласс	**Назначение**	**Пример применения**
:first-child	Выбирает первый дочерний элемент внутри родителя	Выделение первого пункта списка
:last-child	Выбирает последний дочерний элемент внутри родителя	Удаление нижней границы у последнего элемента списка
:nth-child(n)	Выбирает элемент, являющийся n-м дочерним элементом своего родителя	Создание чередующихся строк («зебры») в таблицах или списках

```
/* Выделим первую и последнюю статью в ленте */
.post-list article:first-child {
   border-top: none;
}
.post-list article:last-child {
   border-bottom: none;
}

/* Красим четные строки таблицы */
tr:nth-child(even) {
   background-color: #f2f2f2;
```

}

3. Псевдоклассы ссылок

:link — стили для непосещенных ссылок:

:visited — стили для ссылок, которые пользователь уже посещал.

Псевдоэлементы

Псевдоэлементы позволяют создавать и стилизовать виртуальные элементы, которых нет в HTML-коде. Они используются для оформления определенной части существующего элемента или для добавления декоративного контента.

Исторически псевдоэлементы писались с двойным двоеточием ::, чтобы отличать их от псевдоклассов (с одним двоеточием :). Однако для обратной совместимости браузеры поддерживают и одинарное двоеточие для самых распространенных псевдоэлементов (::before, ::after, ::first-line, ::first-letter).

Самые часто используемые псевдоэлементы ::before и ::after . Они позволяют вставлять контент до или после содержимого выбранного элемента. Для их работы обязательно CSS-свойство content. Даже если оно пустое (content: "";), псевдоэлемент будет создан.

Ключевое отличие псевдокласса от псевдоэлемента это то, что псевдокласс описывает состояние существующего элемента, а псевдоэлемент создает новый, виртуальный элемент, который можно стилизовать.

CSS-фреймворки для ускорения и стандартизации разработки

По мере роста сложности веб-интерфейсов разработчики столкнулись с необходимостью решать одни и те же задачи: создавать адаптивные сетки, стилизовать формы, кнопки, навигацию и обеспечивать единый вид в разных браузерах. Чтобы не «изобретать велосипед» для каждого нового проекта, были созданы CSS-фреймворки — это готовые библиотеки предварительно написанных CSS (и часто JavaScript) классов, которые можно использовать для быстрой и стандартизированной верстки.

Основная цель любого фреймворка — ускорить процесс разработки и обеспечить визуальную и функциональную согласованность всех элементов на странице.

Большинство современных CSS-фреймворков предоставляют следующие возможности:

- Сеточная система, которая позволяет быстро создавать сложные, адаптивные макеты на основе системы колонок (например, 12-колоночную сетку). Вместо того чтобы писать собственные Grid-правила, разработчик просто добавляет нужные классы (например, .col-4) к элементам.
- Стилизация типовых элементов. Готовые классы для стилизации кнопок (<button>), форм (<input>, <select>), таблиц, навигационных меню, выпадающих списков и других стандартных HTML-элементов.
- Утилитарные классы. Разработано множество маленьких классов для выполнения одной конкретной задачи. Например, класс .text-center для выравнивания текста по центру или .m-4 для установки отступа со всех сторон. Это позволяет применять стили прямо в HTML, не создавая новые селекторы в CSS-файле.
- Кроссбраузерная совместимость. Фреймворки решают множество проблем с отображением элементов в разных браузерах «из коробки».
- Адаптивные утилиты. Классы для управления видимостью и поведением элементов на разных устройствах (например, .d-none .d-md-block — элемент скрыт на мобильных и виден на средних и больших экранах).

Популярные CSS-фреймворки

- Bootstrap. Самый популярный фреймворк в мире. Отличается огромным набором готовых компонентов, подробной документацией и большим сообществом. Идеален для быстрого прототипирования и создания административных панелей.

- Tailwind CSS. Утилитарно-ориентированный фреймворк. Он не предлагает готовых компонентов (кнопок, карточек), а дает набор «строительных блоков» (утилит), из которых можно собрать любой дизайн. Это обеспечивает максимальную гибкость и уникальность внешнего вида.
- Bulma. Фреймворк, основанный на Flexbox. Он очень прост в освоении, имеет чистый синтаксис и не использует JavaScript для своей работы (взаимодействие реализовано через Data-атрибуты).

Быстрая верстка с помощью фреймворка Bootstrap

Bootstrap, изначально разработанный в Twitter для обеспечения единообразия внутренних инструментов, стал индустриальным стандартом для быстрого создания адаптивных и современных веб-сайтов.

Основная философия Bootstrap — предоставить разработчику набор готовых, протестированных и кроссбраузерных компонентов (кнопок, форм, навигации, карточек) и мощную сеточную систему, чтобы свести к минимуму написание CSS-кода с нуля.

Самый простой и быстрый способ начать работу — использовать компилированную версию Bootstrap через CDN. Это не требует установки или настройки сборщиков, достаточно добавить несколько строк в ваш HTML-файл.

Вставьте следующий код в секцию <head> вашего HTML-документа, перед подключением ваших собственных стилей, чтобы их можно было переопределять. Обратите внимание: Bootstrap требует jQuery и Popper.js для работы интерактивных компонентов (вроде выпадающих меню, модальных окон). В версии 5.x они подключаются отдельно, как показано ниже.

```
<!DOCTYPE html>
<html lang="ru">
<head>
   <meta charset="UTF-8">
   <title>Проект на Bootstrap</title>
   <!-- Подключение Bootstrap CSS -->
   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
rel="stylesheet">
```

```
    </head>
    <body>

        <!-- Ваш контент здесь -->

        <!-- Подключение необходимых JavaScript-библиотек -->
        <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.8/dist/umd/popper.min.js"></script>
        <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.min.js"></script>
    </body>
    </html>
```

Фундамент любого макета на Bootstrap — это его 12-колоночная адаптивная сетка. Она позволяет легко создавать сложные макеты, которые корректно отображаются на устройствах с любой шириной экрана.

Основные классы сетки

Контейнеры (.container, .container-fluid). Класс .container создает адаптивный контейнер с фиксированной максимальной шириной на разных брейкпоинтах (размерах экрана). Класс .container-fluid создает контейнер на всю ширину окна браузера.

Строки (.row). Строки — это горизонтальные группы колонок. Они необходимы для создания структуры сетки и правильного выравнивания отрицательных отступов.

Колонки (.col) являются основными строительными блоками. Классы колонок создаются для каждого брейкпоинта:

.col- (очень маленькие устройства, <576px)

.col-sm- (маленькие, ≥576px)

.col-md- (средние, ≥768px)

.col-lg- (большие, ≥992px)

.col-xl- (очень большие, ≥1200px)

Пример создания простого макета из двух колонок. В этом примере на мобильных устройствах (col-) колонки будут располагаться одна под другой (по 100% ширины). На экранах от 768px (md) и шире они встанут в один ряд: основная часть займет 2/3 ширины, а сайдбар — 1/3.

```
<div class="container">
  <div class="row">
    <!-- Колонка шириной 8 из 12 на средних и больших экранах, полная ширина на маленьких -->
    <div class="col-md-8">
      <h2>Основной контент</h2>
      <p>Эта колонка занимает 8/12 сетки на экранах от 768px и шире.</p>
    </div>

    <!-- Колонка шириной 4 из 12 на средних и больших экранах, полная ширина на маленьких -->
    <div class="col-md-4">
      <h2>Сайдбар</h2>
      <p>Эта колонка занимает 4/12 сетки на экранах от 768px и шире.</p>
    </div>
  </div>
</div>
```

Преимуществом фреймворка является наличие большого набора готовых компонентов пользовательского интерфейса. Их использование позволяет значительно сократить время разработки веб-страниц и обеспечить единообразное оформление элементов интерфейса без необходимости создания каждого компонента с нуля с использованием CSS.

К числу наиболее часто применяемых компонентов Bootstrap относятся кнопки, навигационные панели и карточки.

Кнопки используются для выполнения пользователем определённых действий: отправки формы, перехода на другую страницу, открытия дополнительной информации и т. д. Для создания кнопки применяется класс .btn, который обычно дополняется классом, определяющим её визуальный вариант. Например, класс .btn-primary используется для основной кнопки, а .btn-success — для кнопки, связанной с успешным или положительным действием.

Пример использования:

```
<button class="btn btn-primary">Сохранить</button>
<button class="btn btn-success">Подтвердить</button>
```

Навигационная панель (.navbar) предназначена для размещения элементов навигации сайта: логотипа, ссылок на основные разделы, кнопок и других элементов управления. Bootstrap предоставляет готовые средства для создания адаптивной навигации, которая автоматически изменяет своё представление в зависимости от размера экрана. Это особенно важно при разработке сайтов, предназначенных для просмотра как на персональных компьютерах, так и на мобильных устройствах.

Карточки (.card) представляют собой универсальный компонент для структурированного отображения информации. Они могут использоваться для представления статей, товаров, новостей, профилей пользователей, результатов поиска и других объектов. Карточка позволяет объединить заголовок, текст, изображение, ссылки и кнопки в едином визуальном блоке.

Помимо готовых компонентов, Bootstrap содержит большое количество утилитарных классов. Такие классы предназначены для быстрого применения отдельных свойств оформления непосредственно к HTML-элементам. Их использование во многих случаях позволяет сократить объём собственного CSS-кода.

Например, для управления внешними и внутренними отступами используются классы:

- .m-3 — задаёт внешний отступ (margin);
- .p-2 — задаёт внутренний отступ (padding).

Числовая часть класса определяет величину отступа в соответствии со шкалой Bootstrap, обычно от 0 до 5. Таким образом, разработчик может быстро подобрать необходимый уровень отступа без создания дополнительного CSS-правила.

Для изменения цвета текста используются классы, начинающиеся с .text-. Например:

- .text-primary — основной цвет текста;
- .text-danger — цвет, используемый для обозначения опасности, ошибки или предупреждения.

Для задания цвета фона применяются классы .bg-*. Например:

- .bg-dark — тёмный фон;
- .bg-success — фон, соответствующий цветовой схеме успешного действия.

Для добавления визуальных эффектов также предусмотрены специальные классы. Например, класс .shadow-sm добавляет элементу небольшую тень.

Таким образом, применение компонентов и утилитарных классов Bootstrap позволяет формировать интерфейс непосредственно средствами HTML, используя заранее определённые правила оформления. Это существенно ускоряет разработку, способствует единообразию интерфейса и упрощает создание адаптивных веб-страниц.

При этом Bootstrap не исключает использование собственного CSS. При необходимости стандартные компоненты и классы могут быть дополнены пользовательскими стилями, что позволяет адаптировать внешний вид сайта под конкретные требования проекта.

Основы языка программирования JavaScript

JavaScript (JS) является языком программирования, неотъемлемой частью современной веб-разработки, отвечая за поведение и логику. Это единственный язык, который нативно (без дополнительных плагинов) исполняется в браузере

пользователя, что делает его главным инструментом для создания интерактивных и динамических веб-приложений.

Язык программирования JavaScript превращает статичную веб-страницу в живой и отзывчивый интерфейс. Его основная задача — обеспечить взаимодействие с пользователем. С помощью JS страница может мгновенно реагировать на действия посетителя: обрабатывать клики по кнопкам, проверять правильность заполнения форм в реальном времени и показывать всплывающие уведомления об успехе или ошибке.

Кроме того, JavaScript позволяет динамически изменять содержимое и внешний вид страницы без её перезагрузки. Это означает, что можно на лету добавлять новые комментарии, удалять товары из корзины, переключать цветовую тему сайта с темной на светлую или показывать и скрывать выпадающие меню и вкладки.

Для улучшения визуального восприятия JS используется для создания анимаций и сложных визуальных эффектов. Хотя простые анимации лучше реализовывать на CSS, именно JavaScript незаменим для создания интерактивных слайдеров, эффектов при прокрутке страницы и динамических графиков.

Одной из самых мощных возможностей JavaScript является работа с данными. С помощью технологии AJAX страница может обмениваться информацией с сервером в фоновом режиме. Именно так работают современные веб-приложения: новостные ленты подгружают новые посты, а чаты обновляют сообщения без перезагрузки всей страницы. Также JS позволяет взаимодействовать с внешними сервисами через их API, например, для отображения прогноза погоды или курса валют.

Наконец, JavaScript предоставляет инструменты для работы с мультимедиа (создание кастомных плееров) и для сохранения данных прямо в браузере пользователя (LocalStorage). Это позволяет запоминать настройки сайта или содержимое корзины даже после закрытия вкладки. В совокупности все эти возможности позволяют создавать сложные и быстрые веб-приложения,

которые по своему поведению ничем не уступают обычным компьютерным программам.

Несмотря на свою мощь и универсальность, JavaScript как язык и технология имеет ряд фундаментальных ограничений, понимание которых критически важно для любого веб-разработчика. Эти ограничения напрямую влияют на архитектуру, безопасность и производительность приложений.

В первую очередь, работа JavaScript строго ограничена моделью безопасности браузера, известной как «песочница» (sandbox). Это намеренное ограничение изолирует код от операционной системы пользователя. Так, скрипт не может получить доступ к локальной файловой системе для чтения или изменения файлов, что защищает личные данные от кражи. Кроме того, браузер подчиняется политике единого источника, которая запрещает скриптам с одного сайта отправлять запросы к данным на другом сайте, предотвращая несанкционированный доступ к кукам и другой конфиденциальной информации.

Другим важным аспектом является производительность. JavaScript в браузере является однопоточным, что означает выполнение только одной операции в конкретный момент времени. Если скрипт выполняет ресурсоёмкое вычисление, он блокирует основной поток, из-за чего страница перестаёт отвечать на действия пользователя, а анимации замирают. Для решения этой проблемы в языке были реализованы механизмы асинхронности и веб-воркеры, позволяющие выполнять тяжёлые задачи в фоновом режиме.

Также стоит учитывать зависимость JavaScript от окружения. Код выполняется внутри движка конкретного браузера, и различия в их реализации могут приводить к тому, что приложение работает некорректно в разных браузерах. Кроме того, пользователь может отключить выполнение скриптов, что полностью лишит интерактивный сайт его функциональности. Поэтому при разработке важно следовать принципу прогрессивного улучшения, обеспечивая доступность основного контента даже без JavaScript.

Наконец, необходимо помнить о безопасности на стороне клиента. Любой код, отправляемый пользователю, полностью ему виден, поэтому на JavaScript нельзя реализовывать секретную логику (например, проверку лицензионных ключей). Все критически важные проверки должны дублироваться на сервере. Неправильно написанный код также может стать причиной уязвимостей, таких как межсайтовый скриптинг, когда злоумышленник внедряет вредоносный скрипт на страницу.

В этом разделе мы рассмотрим основы синтаксиса языка и его ключевую роль во фронтенд-разработке — взаимодействие с объектной моделью документа (DOM).

Основы синтаксиса JavaScript

Синтаксис JavaScript во многом похож на другие языки программирования семейства C (такие как C++, Java, C#). Код выполняется последовательно, строка за строкой.

Ниже в виде примера приведен код вывода модального окна с помощью функции alert.

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Приветствие через Alert</title>
</head>
<body>

    <h1>Страница с приветствием</h1>
    <p>При загрузке этой страницы сработает встроенное модальное окно браузера</p>

    <script>
        // --- Логика на JavaScript ---

        /* Функция alert() выводит модальное окно с текстом и кнопкой "ОК"
        Код внутри тега <script> выполняется сразу после загрузки страницы */
        alert("Привет, посетитель!\n\nДобро пожаловать на наш сайт. Приятного просмотра!");

        // Примечание: \n создает перенос строки внутри текста окна.
    </script>

</body>
</html>
```

В HTML страницу код JavaScript в примере внедрен с помощью метода внутреннего скрипта, с помощью тега <script></script> (таблица 11). Каждая инструкция заканчивается точкой запятой. Комментарии однострочные начинаются с двойной косой черты //, многострочные – ограничиваются знаками /*…*/.

Таблица 11 — Способы подключения JavaScript к HTML-документу

Метод	**Как выглядит**	**Когда использовать**
внешний файл	<script src="file.js"></script>	Всегда, когда это возможно. Это стандарт для реальных проектов.
внутренний скрипт	<script> ...код... </script>	Для небольших скриптов на одной странице или для быстрого прототипирования.
инлайновые атрибуты	<button onclick="...">	Лучше избегать. Подходит только для самых простых учебных примеров.

Теги <script> (как для внешних, так и для внутренних скриптов) обычно размещают в самом конце тега <body>. Это гарантирует, что вся HTML-страница загрузится первой, и JavaScript-код сможет без ошибок найти и изменить нужные элементы.

Переменные и типы данных

Для хранения данных используются переменные. В современном JavaScript (стандарт ES6+) для их объявления предпочтительно использовать ключевые слова let и const. Ключевое слово let объявляет переменную, значение которой может быть изменено в дальнейшем, а const объявляет константу. Присвоить ей значение можно только один раз.

JavaScript относится к языкам с динамической нестрогой типизацией. Это одна из его ключевых особенностей, которая одновременно делает его очень гибким и в то же время является источником множества ошибок. Тип определяется автоматически в момент присваивания значения. Более того, вы можете в любой момент изменить тип данных, хранящихся в переменной.

В JavaScript существует несколько основных (примитивных) типов данных, которые используются для хранения информации в переменных. Понимание этих типов необходимо для написания корректного кода.

Число (number) используется для представления как целых, так и дробных чисел. В JavaScript нет отдельного типа для целых чисел (Integer) и чисел с плавающей точкой (Float), есть только Number. Тип Number также включает в себя специальные значения Infinity (бесконечность) и NaN (Not-a-Number).

В JavaScript тип переменной определяется в момент присваивания ей значения. Проверить тип можно с помощью оператора typeof.

```
let age = 30; // Целое число
let price = 9.99; // Дробное число
let temperature = -5; // Отрицательное число

console.log(typeof age); // "number"

// Специальные значения
let divisionByZero = 10 / 0;
console.log(divisionByZero); // Infinity

let invalidCalc = "текст" / 2;
console.log(invalidCalc); // NaN
console.log(typeof invalidCalc); // "number"
```

Строка (string) представляет собой последовательность символов. Строки в JavaScript можно заключать в одинарные ('...'), двойные ("...") или обратные кавычки (`...`). Примеры: 'Привет', "Мир", `Цена: ${price} руб.`. В JavaScript для конкатенации строк используется оператор + (плюс).

Когда оператор + применяется к строкам, он не складывает их как числа, а склеивает их друг за другом. Если вы попытаетесь объединить строку с числом, JavaScript автоматически преобразует число в строку и выполнит склеивание.

Современный и более удобный способ объединения строк и переменных — использование шаблонных строк. Они заключаются в обратные кавычки (`...`). Строки в обратных кавычках позволяют легко вставлять значения переменных прямо в текст с помощью конструкции ${...}.

```
let name = "Алексей";
let age = 25;
```

```
// конкатенация
let messageKonkatenaziya = "Меня зовут " + name + ". Мне " + age + " лет.";
console.log(messageKonkatenaziya); // "Меня зовут Алексей. Мне 25 лет."

// шаблонные строки
let messageTemplate = `Меня зовут ${name}. Мне ${age} лет.`;
console.log(messageTemplate); // "Меня зовут Алексей. Мне 25 лет."
```

В JavaScript строки это один из основных типов данных, представляющих последовательность символов. Строки в JavaScript являются неизменяемыми объектами, что означает, что после создания строки её нельзя изменить. Однако можно создавать новые строки на основе существующих.

Основные свойства и методы работы со строками в JavaScript предоставлены в таблице 12.

Таблица 12 — Свойства и методы работы со строками в JavaScript

Свойство/метод	**Описание**	**Пример**
length	возвращает количество символов в строке	let str = "Hello"; console.log(str.length); // 5
charAt(index)	возвращает символ по указанному индексу	let str = "Hello"; console.log(str.charAt(0)); // "H"
[]	альтернативный способ получения символа по индексу	let str = "Hello"; console.log(str[1]); // "e"
indexOf(substring)	ищет первое вхождение подстроки	let str = "Hello, world!"; console.log(str.indexOf("world")); // 7
lastIndexOf(substring)	ищет последнее вхождение подстроки	let str = "Hello, world!"; console.log(str.lastIndexOf("o")); // 8
slice(start, end)	извлекает часть строки	let str = "Hello, world!"; console.log(str.slice(0, 5)); // "Hello"
substring(start, end)	аналог slice, но не допускает отрицательные значения	let str = "Hello, world!"; console.log(str.substring(7, 12)); // "world"
substr(start, length)	извлекает часть строки длиной length, начиная с позиции start	let str = "Hello, world!"; console.log(str.substr(7, 5)); // "world"
replace(oldsubstring, newsubstring)	заменяет первое вхождение oldsubstring на newsubstring	let str = "Hello, world!"; console.log(str.replace("world", "Earth")); // "Hello, Earth!"
toLowerCase()	преобразует строку в нижний регистр	let str = "Hello, world!"; console.log(str.toLowerCase()); // "hello, world!"
toUpperCase()	преобразует строку в верхний	let str = "Hello, world!";

	регистр	console.log(str.toUpperCase()); // "HELLO, WORLD!"
trim()	удаляет пробельные символы в начале и конце строки	let str = " Hello, world! "; console.log(str.trim()); // "Hello, world!"
split(separator)	разбивает строку на массив подстрок по указанному разделителю	let str = "apple,banana,orange"; console.log(str.split(",")); // ["apple", "banana", "orange"]
concat(string1, string2, ...)	объединяет несколько строк в одну	let str1 = "Hello"; let str2 = ", world!"; console.log(str1.concat(str2)); // "Hello, world!"
includes(substring)	проверяет, входит ли подстрока в строку	let str = "Hello, world!"; console.log(str.includes("world")); // true
repeat(count)	повторяет строку указанное количество раз	let str = "Hi"; console.log(str.repeat(3)); // "HiHiHi"

Логический, булев тип (boolean) имеет всего два возможных значения: true (истина) и false (ложь). Он используется для управления логикой программы — в условных операторах (if/else) и циклах.

```
let isLoggedIn = true;
let hasPermission = false;

if (isLoggedIn) {
   console.log("Пользователь вошел в систему.");
} else {
   console.log("Пользователь не вошел.");
}
```

Нулевое значение (null) является примитивным значением, которое представляет собой намеренное отсутствие какого-либо значения. Это не ошибка, а способ явно сказать: «Здесь должно быть что-то, но сейчас это пусто».

```
let user = { name: "Иван", age: 25 };
user = null; // Мы явно указываем, что переменная user больше ничего не содержит
```

Неопределено (undefined) является типом, который означает, что значение не было присвоено. Это значение по умолчанию для объявленных, но неинициализированных переменных, когда функция не возвращает значение явно (возвращает undefined по умолчанию) и когда вы пытаетесь обратиться к

свойству объекта, которого не существует. Ключевое отличие от нулевого значения в том, что null присваивается намеренно разработчиком, а undefined присваивается автоматически интерпретатором JavaScript.

```
let age; // Объявили переменную, но не присвоили значение
console.log(age); // Выведет: undefined

function doNothing() {
   // Код, который ничего не возвращает
}
let result = doNothing();
console.log(result); // Выведет: undefined
```

Символ (symbol) является уникальным и неизменяемым типом данных, который часто используется как идентификатор для свойств объектов. Каждый вызов Symbol() создает уникальное значение, даже если передать ему одно и то же описание. Используется для создания «скрытых» или уникальных свойств объектов, чтобы избежать случайного перезаписывания данных.

```
let id1 = Symbol("id");
let id2 = Symbol("id");

console.log(id1 === id2); // false! Символы всегда уникальны

// Используем символ как ключ в объекте
const user = {
   name: "Вася",
   [id1]: 123 // Свойство с уникальным ключом-символом
};
console.log(user[id1]); // 123
console.log(user.id1);  // undefined (такого свойства нет)
```

Большое целое число (BigInt) — это числовой тип для представления целых чисел произвольной длины. Обычный тип Number в JavaScript не может безопасно работать с числами больше (2^{53} – 1). Используется, например, для финансовых расчетов с очень большими суммами или в криптографии, где требуются огромные целые числа.

```
// Обычное число теряет точность
console.log(9007199254740991 + 1000); // 9007199254741992 (неверно!)

// BigInt сохраняет точность
const bigNumber = 9007199254740991n;
console.log(bigNumber + 1000n); // 9007199254741991n (верно!)
```

```
// Важное правило: нельзя смешивать Number и BigInt напрямую
// console.log(10 + 10n); // Ошибка! Нужно приводить к одному типу
console.log(10 + Number(10n)); // 20
```

Тип данных Date в JavaScript играет ключевую роль в работе с датами и временем, предоставляя богатый набор возможностей для создания, изменения и форматирования календарных дат и временных промежутков.

Создание объекта Date возможно несколькими способами. Конструктор без параметров (new Date()) — создает объект с текущей датой и временем. Конструктор с параметрами (new Date(year, month, day, hours, minutes, seconds, milliseconds)) — позволяет задать точную дату и время. Конструктор со строкой (new Date("January 5, 2020")) — создает объект на основе строки, представляющей дату и время.

Тип Date предоставляет множество полезных методов для извлечения и изменения составляющих даты:

getFullYear(), getMonth(), getDay(), getHours(), getMinutes(), getSeconds() — позволяют получить составляющие даты и времени;

setFullYear(), setMonth(), setHours() и другие методы — позволяют изменять значения даты и времени;

getTime() — возвращает количество миллисекунд, прошедших с 1 января 1970 года (Unix epoch time).

По умолчанию объекты Date работают с UTC (Universal Coordinated Time), но методы могут возвращать значения с учётом местного часового пояса. Это важно учитывать при международной разработке, так как методы могут возвращать разное время в зависимости от локального часового пояса. Для форматирования даты и времени часто используются дополнительные библиотеки, такие как Moment.js или Luxon, которые предоставляют более гибкие возможности. Однако в современных версиях JavaScript есть встроенные методы, такие как toLocaleDateString() и toLocaleTimeString(), которые позволяют выводить дату и время в удобочитаемом формате, учитывая

региональные настройки. Также следует помнить, что в JavaScript месяцы в конструкторе Date начинаются с 0 (январь) и заканчиваются на 11 (декабрь).

Операторы в JavaScript

В JavaScript, как и в любом другом языке программирования существуют операторы, специальные символы (или ключевые слова), которые выполняют операции над одним, двумя или тремя операндами.

Операнд — это значение или переменная, над которой совершается действие.

Все операторы в JS можно разделить на несколько основных категорий.

1. Арифметические операторы, которые используются для выполнения математических вычислений.

Сложение (+) используется для суммирования чисел или же выполняет конкатенацию строк.

10 + 5 даст 15
'Привет, ' + 'мир!' даст 'Привет, мир!'

Вычитание (-) используется для того, чтобы вычесть одно число из другого.

10 - 5 даст 5

Умножение (*) используется для перемножения чисел.

10 * 5 даст 50

Деление (/) используется для деления одного числа на другое.

10 / 5 даст 2

Остаток от деления (%) возвращает остаток от деления (модуль).

12 % 5 даст 2 (потому что 12 = 2 * 5 + 2)

Возведение в степень (**) возводит число в степень.

2 ** 3 даст 8 (2 в кубе)

Инкремент (++) увеличивает значение переменной на 1.

Если let x = 5;, то x++ сделает x равным 6.

Декремент (--) уменьшает значение переменной на 1.

Если let y = 5;, то y-- сделает y равным 4.

2. Операторы присваивания используются для присваивания значений переменным.

Присваивание (=) присваивает значение справа переменной слева.

let a = 10;

Сложное присваивание или комбинация арифметического оператора и присваивания.

a += 5; — то же самое, что a = a + 5; (результат: 15)

a -= 3; — то же самое, что a = a - 3; (результат: 12)

a *= 2; — то же самое, что a = a * 2; (результат: 24)

3. Операторы сравнения (таблица 13) используются для сравнения значений и всегда возвращают логический результат: true или false, чаще всего используются в условных конструкциях (if, циклы).

Таблица 13 — Операторы сравнения в JavaScript

Оператор	**Описание**	**Пример**
равно (==)	проверяет равенство значений с приведением типов (менее строгий)	5 == '5' вернет true
строго равно (===)	проверяет равенство без приведения типов (строгий). сравнивает и тип, и значение	5 === '5' вернет false
не равно (!=)	проверяет неравенство значений с приведением типов	5 != '5' вернет false
строго не равно (!==)	проверяет неравенство без приведения типов. сравнивает и тип, и значение	5 !== '5' вернет true
больше (>)	возвращает true, если левый операнд строго больше правого	10 > 5 вернет true
меньше (<)	возвращает true, если левый операнд строго меньше правого	10 < 5 вернет false
больше или равно (>=)	возвращает true, если левый операнд больше или равен правому	10 >= 10 вернет true
меньше или равно (<=)	возвращает true, если левый операнд меньше или равен правому	5 <= 10 вернет true

4. Логические операторы (таблица 14) работают с логическими значениями (true, false) и позволяют строить сложные условия.

Таблица 14 — Логические операторы в JavaScript

Оператор	**Описание**	**Пример**
и (&&)	возвращает true, только если оба операнда истинны. в противном случае возвращает false	(5 > 3 && 10 < 20) вернет true
или (||)	возвращает true, если хотя бы один из	(5 > 10 || 10 < 20) вернет true

	операндов истинен	
не (!)	инвертирует (меняет на противоположное) логическое значение операнда. превращает true в false, а false в true	!true вернет false

5. Оператор typeof является унарным оператором, который возвращает строку, указывающую на тип данных операнда. Очень полезен для отладки.

```
typeof 42;         // "number"
typeof "текст";    // "string"
typeof true;       // "boolean"
typeof undefined;  // "undefined"
typeof null;       // "object" (известная ошибка в JS)
typeof {};         // "object"
typeof [];         // "object"
typeof function(){} // "function"
```

Циклы и условные конструкции

Циклы и условные конструкции — это важнейшие элементы любого языка программирования, включая JavaScript. Они позволяют организовывать выполнение кода в зависимости от условий и повторяющихся действий. Рассмотрим основные виды циклов и конструкций ветвления в JavaScript.

Условная конструкция if...else является самой базовой конструкцией ветвления. Выполняет блок кода, если условие истинно, и другой блок, если условие ложно.

```
if (condition) {
   // Код, который выполняется, если condition истинно
} else {
   // Код, который выполняется, если condition ложно
}
```

Конструкция switch-case является альтернативой множественным условиям if...else. Удобен для выбора из нескольких вариантов.

```
switch (expression) {
   case value1:
      // Код, если expression равно value1
```

```
        break;
    case value2:
        // Код, если expression равно value2
        break;
    default:
        // Код, если не совпал ни один case
}
```

Цикл while выполняет блок кода, пока условие истинно.

```
while (condition) {
    // Код, который выполняется, пока condition истинно
}
```

Цикл do...while сначала выполняет блок кода, а затем проверяет условие.

```
do {
    // Код, который выполняется минимум один раз
} while (condition);
```

Классический цикл for с тремя частями: инициализация, условие и инкремент.

```
for (initialization; condition; final-expression) {
    // Код, который выполняется на каждой итерации
}
for(let i = 0; i < 5; i++) {
    console.log(i);
}
```

Цикл for...in проходит по свойствам объекта.

```
for (variable in object) {
    // Код, который выполняется для каждого свойства объекта
}
```

Цикл for...of проходит по элементам итерируемых объектов (массивы, строки и т.д.).

```
for (variable of iterable) {
    // Код, который выполняется для каждого элемента
}
```

Циклы и условные конструкции являются инструментами для управления потоком выполнения программы, они позволяют автоматизировать повторяющиеся действия и принимать решения на основе условий.

Объекты и массивы в JavaScript.

Объекты и массивы — это два самых важных и часто используемых типа данных в JavaScript. По сути, это способы хранения коллекций данных, но они работают по-разному и используются для разных целей.

В основе обоих лежит концепция "объект". В JavaScript массив — это тоже объект, но со специальным поведением.

Объект (objects) это неупорядоченная коллекция пар "ключ: значение", используемый для описания сущности с набором свойств. Ключ — это всегда строка (или Symbol), а значение может быть чем угодно: числом, строкой, другой функцией или даже другим объектом.

Литеральная нотация:

```
let user = {
  name: "Иван",
  age: 30,
  isAdmin: true,
  sayHello: function() {
    console.log("Привет!");
  }
};

// Доступ к свойствам через точку
console.log(user.name); // "Иван"
user.sayHello(); // "Привет!"

// Доступ через квадратные скобки (полезно, если ключ — переменная)
let key = "age";
console.log(user[key]); // 30
```

Изменение и добавление свойств. Вы можете легко добавлять, изменять и удалять свойства после создания объекта.

```
user.age = 31; // Изменили возраст
user.city = "Москва"; // Добавили новое свойство
delete user.isAdmin; // Удалили свойство
```

Массив (Arrays) — это упорядоченная коллекция значений, идеально подходящая для хранения списков (например, список товаров, список пользователей). В JavaScript массив — это объект, у которого ключи являются числовыми индексами (0, 1, 2 и т.д.). Значения перечисляются внутри квадратных скобок [] через запятую.

```
let fruits = ["Яблоко", "Банан", "Апельсин"];

// Доступ к элементам по индексу (нумерация с нуля!)
console.log(fruits[0]); // "Яблоко"
console.log(fruits[1]); // "Банан"
```

Основные методы работы с массивами

Встроенные методы для работы с массивами можно разделить на несколько групп:

1. Методы для добавления и удаления элементов изменяют исходный массив.

Метод push(item) добавляет один или несколько элементов в **конец** массива и возвращает новую длину.

```
let fruits = ["Яблоко", "Банан"];
fruits.push("Апельсин"); // Добавляем в конец
console.log(fruits); // ["Яблоко", "Банан", "Апельсин"]
```

Метод pop() удаляет последний элемент из массива и возвращает его.

```
let fruits = ["Яблоко", "Банан", "Апельсин"];
let last = fruits.pop();
console.log(last); // "Апельсин"
console.log(fruits); // ["Яблоко", "Банан"]
```

Метод shift() удаляет первый элемент из массива и возвращает его. Работает медленно на больших массивах, так как нужно переиндексировать все элементы.

```
let fruits = ["Яблоко", "Банан", "Апельсин"];
let first = fruits.shift();
console.log(first); // "Яблоко"
console.log(fruits); // ["Банан", "Апельсин"]
```

Метод unshift(...items) добавляет один или несколько элементов в начало массива и возвращает новую длину.

```
let fruits = ["Банан", "Апельсин"];
fruits.unshift("Яблоко");
console.log(fruits); // ["Яблоко", "Банан", "Апельсин"]
```

2.Методы для поиска в массиве

Метод indexOf(item, from) ищет элемент item начиная с индекса from и возвращает его индекс, или -1, если ничего не найдено.

```
let fruits = ["Яблоко", "Банан", "Апельсин"];
console.log(fruits.indexOf("Банан")); // 1
console.log(fruits.indexOf("Груша")); // -1 (не найдено)
```

Метод includes(value) проверяет, есть ли в массиве элемент со значением value. Возвращает true или false.

```
let fruits = ["Яблоко", "Банан"];
console.log(fruits.includes("Банан")); // true
console.log(fruits.includes("Груша")); // false
```

Метод find(callback) — ищет первый элемент, для которого функция callback вернет true. Возвращает сам элемент (или undefined).Отлично подходит для поиска объекта в массиве объектов.

```
let users = [
  {id: 1, name: "Иван"},
  {id: 2, name: "Петр"}
];

let user = users.find(item => item.id === 2);
console.log(user.name); // "Петр"
```

3. Методы для трансформации массива

Важной особенностью этих методов является то, что они не изменяют исходный массив, они всегда возвращают новый массив.

Метод map(callback) создает новый массив, вызывая функцию callback для каждого элемента исходного массива. Результат каждого вызова добавляется в новый массив, этот метод чаще всего используется дляизменения данных.

```
let lengths = ["Яблоко", "Груша", "Вишня"].map(item => item.length);
console.log(lengths); // [6, 4, 5]
```

Метод filter(callback) создает новый массив, включая в него только те элементы исходного массива, для которых функция callback вернула true. Используется для отбора (фильтрации) данных.

```
let numbers = [1, 2, 3, 4, 5];
```

```
let evenNumbers = numbers.filter(item => item % 2 === 0);
console.log(evenNumbers); // [2, 4]
```

Метод sort(compareFunction) сортирует исходный массив, без функции сравнения сортирует элементы как строки (что приводит к странному порядку чисел: 1, 10, 2), поэтому необходимо передавать функцию сравнения для правильной сортировки чисел.

```
let numbers = [1, 10, 21, 2];

// Неправильная сортировка (как строки)
numbers.sort();
console.log(numbers); // [1, 10, 2, 21]

// Правильная сортировка (как числа)
numbers.sort((a, b) => a - b);
console.log(numbers); // [1, 2, 10, 21]
```

4. Методы для перебора (итерации)

Метод forEach(callback) выполняет функцию callback для каждого элемента массива, не возвращает ничего и используется вместо цикла for, когда не нужно создавать новый массив.

```
["Яблоко", "Банан"].forEach((item, index) => {
  console.log(`${index}: ${item}`);
});
```

5. Методы для сведения массива к одному значению

Метод reduce(callback, initialValue) "сворачивает" (редуцирует) массив в одно единственное значение (например, сумму всех элементов).Функция callback принимает два аргумента: аккумулятор (acc) и текущий элемент (item).

```
let numbers = [1, 2, 3, 4];

// Суммируем все числа
let sum = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
}, 0); // Второй аргумент (0) - начальное значение аккумулятора

console.log(sum); // 10 (0+1+2+3+4)

// Сцепим все строки в одну
let words = ["Я", "изучаю", "JavaScript"];
let sentence = words.reduce((acc, word) => acc + " " + word);
```

```
console.log(sentence); // " Я изучаю JavaScript"
```

Для взаимодействия с пользователем через диалоговые окна в JavaScript существуют встроенные методы alert(), prompt() и confirm(). Они просты в использовании и позволяют быстро получать реакцию от пользователя или информировать его о чём-либо. Рассмотрим каждую из них подробнее.

1. Функция alert() отображает модальное окно с сообщением и единственной кнопкой «ОК». Она используется для вывода коротких уведомлений или предупреждений.

```
alert("Привет, пользователь!");
```

2. Функция prompt() отображает модальное окно с полем для ввода текста и двумя кнопками: «ОК» и «Отмена». Она позволяет запрашивать у пользователя короткие текстовые данные.

```
let name = prompt("Введите ваше имя:", "Гость");
console.log(name); // выведет введённое имя или null, если пользователь нажал "Отмена"
```

3.Функция confirm() отображает модальное окно с вопросом и двумя кнопками: «ОК» и «Отмена». Она используется для получения бинарного ответа от пользователя (да/нет).

```
let result = confirm("Вы уверены?");
console.log(result); // true, если нажата "ОК", false, если "Отмена"
```

Все три функции блокируют выполнение кода до тех пор, пока пользователь не примет решение. Это может мешать работе сложных приложений, поэтому эти методы подходят для простых взаимодействий, но для сложных форм и диалогов лучше использовать кастомные решения. Функции поддерживаются всеми современными браузерами, но их внешний вид и поведение могут незначительно отличаться в разных браузерах. Для более сложных и гибких диалогов часто используются библиотеки и фреймворки, такие как Bootstrap Modal, React-модальные окна или Vue.js компоненты. Они позволяют создавать красивые и функциональные диалоговые окна с богатой функциональностью.

Функции в JavaScript

Функция это именованный блок программного кода, который создается один раз, но может выполняться многократно по мере необходимости. В контексте веб-разработки функции являются основным строительным блоком для реализации логики страницы, обработки событий и организации кода. Понимание функций позволяет перейти от простого написания последовательных команд к структурированному и модульному программированию.

В JavaScript существует несколько способов объявления функций, но наиболее распространенным и современным является использование функциональных выражений с ключевым словом const.

```
const имяФункции = function(параметры) {
   // Тело функции: код, который будет выполнен
   console.log("Функция выполняется!");
   return результат; // Опционально: возвращает значение
};
```

Мы создаем переменную и присваиваем ей функцию, при этом использование const является хорошей практикой, так как ссылка на функцию обычно не должна меняться. Ключевое слово function создает объект функции, в скобках указываются параметры — входные данные, которые функция ожидает получить. Код внутри фигурных скобок {тело} выполняется только в момент вызова функции. Оператор return завершает выполнение функции и возвращает результат вызывающему коду. Если return отсутствует или вызван без значения, функция возвращает undefined.

Чтобы запустить код внутри функции, необходимо написать её имя и передать аргументы (если они требуются).

```
// Объявляем функцию
const greetUser = function(userName) {
   console.log("Привет, " + userName + "!");
};

// Вызываем функцию с аргументом "Алексей"
greetUser("Алексей"); // Выведет в консоль: "Привет, Алексей!"
// Вызываем функцию с аргументом "Алексей"
greetUser("Алексей"); // Выведет в консоль: "Привет, Алексей!"
```

Важно различать понятия «параметр» и «аргумент». Параметр — это переменная, указанная в скобках при объявлении функции. Она работает как «заполнитель» внутри функции, аргумент же это конкретное значение, которое передается в функцию в момент её вызова. Функции могут принимать любое количество параметров (включая ноль) и возвращать только одно значение (или undefined).

Современный стандарт ES6 ввел анонимные функции и стрелочные функции (Arrow Functions), которые имеют более короткий синтаксис и лексически привязывают контекст this.

```
// Если тело функции состоит из одной строки, фигурные скобки и return не нужны
const sum = (a, b) => a + b;
console.log(sum(2, 3)); // 5

// Если тело функции содержит несколько строк, нужны фигурные скобки и явный return
const multiply = (x, y) => {
   let result = x * y;
   return result;
};
```

Стрелочные функции особенно удобны при работе с методами массивов (map, filter, reduce) и при передаче колбэков (вспомогательных функций) в другие функции.

В JavaScript функции могут быть свойствами объектов, в этом случае они называются методами. Внутри метода ключевое слово this ссылается на сам объект, что позволяет обращаться к другим его свойствам.

```
let user = {
   name: "Иван",
   // Метод объекта
   sayHello: function() {
      console.log("Привет, меня зовут " + this.name);
   }
};

user.sayHello(); // Вызов метода. Выведет: "Привет, меня зовут Иван"
```

DOM (Document Object Model)

Браузерное окружение является средой выполнения JavaScript-кода в веб-браузере. Оно предоставляет разработчикам доступ к объектам и API, которые

позволяют взаимодействовать с веб-страницами, осуществлять сетевые запросы, работать с графикой и звуком, а также управлять поведением браузера (рис.3).

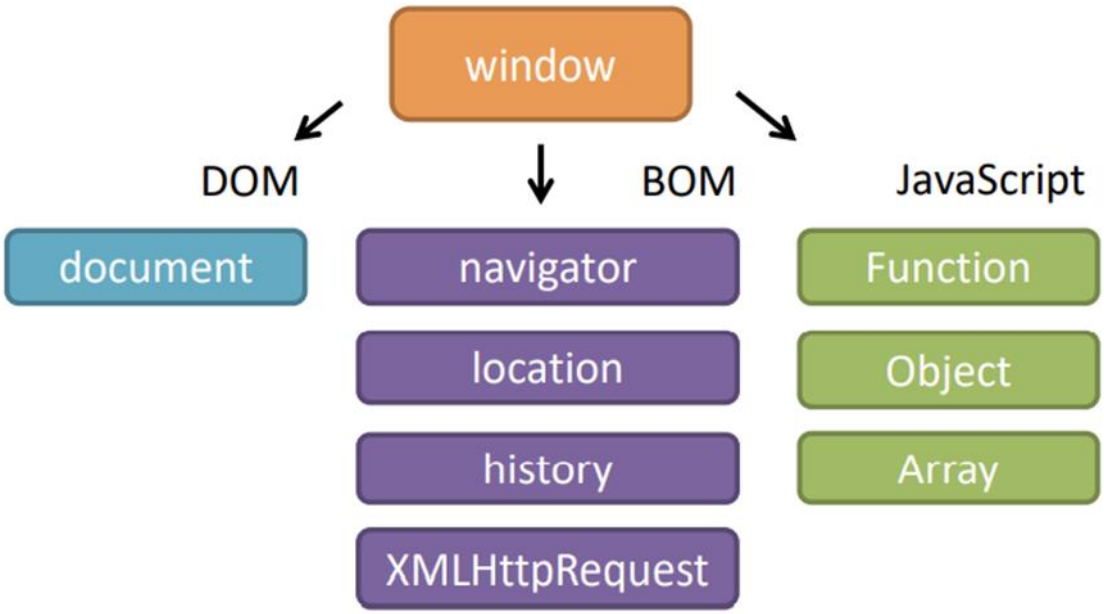


Рисунок 3 - Компоненты браузерного окружения

Основные компоненты браузерного окружения сведены в таблицу 15.

Таблица 15 — Компоненты браузерного окружения (BOM)

Компонент	**Описание**	**Пример использования**
window	глобальный объект, представляющий текущее окно браузера	window.alert("Привет!");
document	объект, представляющий HTML-документ	document.getElementById('element')
location	управляет URL-адресом текущей страницы	window.location.href = 'https://example.com';
history	позволяет управлять историей посещённых страниц	history.back();
таймеры (Timeouts & Intervals)	планирование выполнения кода через определённые интервалы времени	setTimeout(() => {}, 1000);
сетевые запросы (XMLHttpRequest, Fetch API)	отправка HTTP-запросов и получение данных с серверов	fetch('https://api.example.com/data');
каналы коммуникации (WebSockets, EventSource)	поддержка двусторонней связи с сервером в реальном времени	new WebSocket('ws://example.com/ws');

графика и анимация (Canvas, SVG, WebGL)	рисование графики и создание анимаций прямо в браузере	canvas.getContext('2d').fillRect(0, 0, 100, 100);
звук (Audio API)	предоставляет инструменты для работы с аудио-данными	new Audio('sound.mp3').play();
локальное хранилище (LocalStorage, SessionStorage)	позволяет сохранять данные на стороне клиента	localStorage.setItem('key', 'value');
рабочие потоки (Workers)	выполнение тяжёлых вычислений в фоновом режиме, не блокируя основной поток	new Worker('worker.js');

Браузерное окружение предоставляет разработчикам инструменты для создания интерактивных и динамических веб-приложений. Понимая его возможности и ограничения, можно создавать качественные и производительные веб-продукты. Важные аспекты браузерного окружения включают:

– модель событий (event loop) — механизм, который управляет очередью выполнения асинхронных задач. Он обеспечивает правильное чередование синхронного и асинхронного выполнения кода, что критически важно для отзывчивости интерфейсов;

– безопасность (security) — политика, которая ограничивает доступ к ресурсам других доменов (Same-Origin Policy), предотвращая потенциальные угрозы безопасности, такие как атаки Cross-Site Scripting (XSS) и Cross-Site Request Forgery (CSRF);

– производительность (performance) — оптимизация кода и минимизация сетевых запросов для ускорения загрузки страниц. Это включает использование кеширования, уменьшение количества HTTP-запросов, оптимизацию изображений и скриптов, а также правильную организацию асинхронных задач.

DOM (Document Object Model) — это интерфейс, который представляет структуру HTML-документа в виде дерева объектов (рис.4).

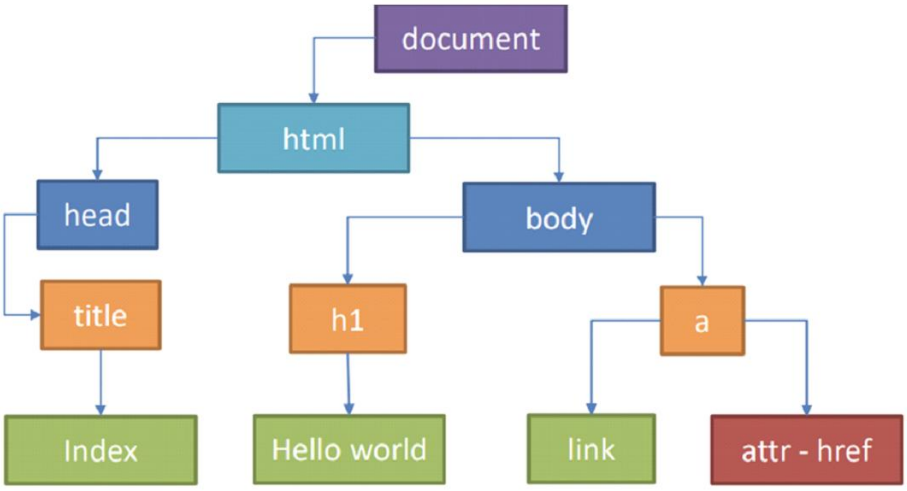


Рисунок 4 - DOM (Document Object Model)

JavaScript позволяет взаимодействовать с DOM, изменяя содержимое страницы, стили и поведение элементов. Рассмотрим основные аспекты работы с DOM в JavaScript.

Для получения элементов DOM используются следующие методы:

– document.getElementById(id) — возвращает элемент по его уникальному идентификатору;

– document.getElementsByTagName(tagName) — возвращает коллекцию элементов с указанным тегом;

– document.getElementsByClassName(className) — возвращает коллекцию элементов с указанным классом;

– document.querySelector(selector) — возвращает первый элемент, соответствующий CSS-селекторам;

– document.querySelectorAll(selector) — возвращает все элементы, соответствующие CSS-селекторам.

```
let header = document.getElementById('header');
let paragraphs = document.getElementsByTagName('p');
let items = document.querySelectorAll('.item');
```

Для навигации между узлами документа DOM для доступа к различным элементам и их свойствам в JavaScript существует несколько способов перемещения по DOM-дереву. Эти свойства позволяют эффективно

перемещаться по DOM-дереву, получая доступ к различным элементам и их отношениям.

– element.childNodes — возвращает список всех дочерних узлов элемента (включая текстовые узлы);

– element.children — возвращает список только элементов (узлов типа Element);

– element.firstChild — возвращает первый дочерний узел;

– element.lastChild — возвращает последний дочерний узел;

– element.nextSibling — возвращает следующий соседний узел;

– element.previousSibling — возвращает предыдущий соседний узел;

– element.parentNode — возвращает родительский узел;

– element.ownerDocument — возвращает объект документа, к которому принадлежит узел.

```
// Получаем элемент по id
let element = document.getElementById('myElement');

// Переходим к первому дочернему узлу
let firstChild = element.firstChild;

// Переходим к последнему дочернему узлу
let lastChild = element.lastChild;

// Переходим к следующему соседнему узлу
let nextSibling = element.nextSibling;

// Переходим к предыдущему соседнему узлу
let previousSibling = element.previousSibling;

// Переходим к родителю
let parent = element.parentNode;

// Переходим к владельцу документа
let ownerDoc = element.ownerDocument;
```

Для создания узлов DOM используются следующие методы:

– document.createElement(tagName) — создание нового элемента, указанного в параметрах;

– document.createTextNode(текст) — создание текстового узла.

Для того, чтобы созданные новые элементы были видимыми, их необходимо вставить в документ. Для этого используются методы append, prepend, before, after, replace (рис.5).

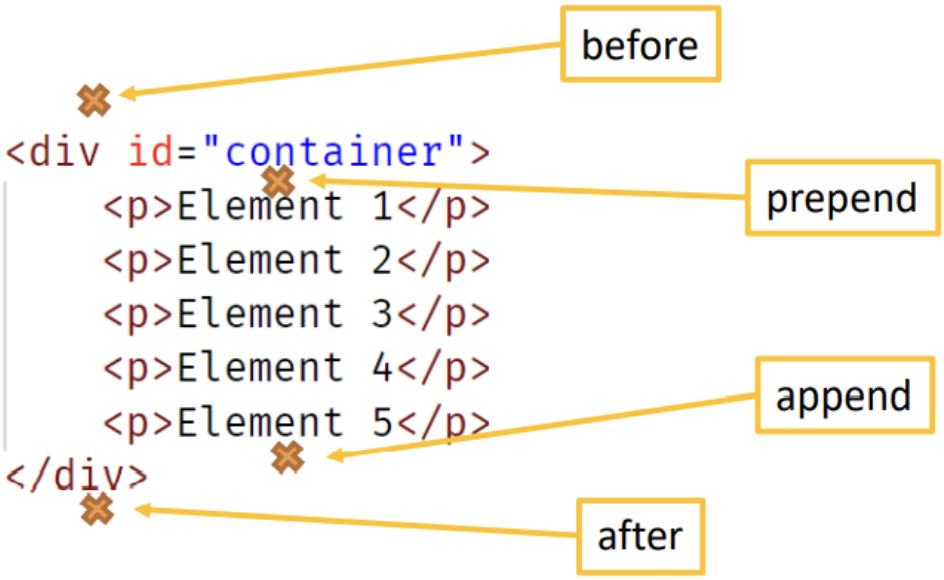


Рисунок 5 - Методы append, prepend, before, after, replace

Для удаления узлов DOM используются следующие методы:

– removeChild(node) — удаляет указанный узел из родительского элемента;

```
let parent = document.getElementById('parent');
let child = document.getElementById('child');
parent.removeChild(child);
```

– remove() — современный метод, который удаляет текущий элемент из DOM;

```
let elementToRemove = document.getElementById('element');
elementToRemove.remove();
```

– innerHTML — установка пустого значения для innerHTML родительского элемента удаляет все дочерние элементы;

```
let parent = document.getElementById('parent');
parent.innerHTML = ''; // Удаляет все дочерние элементы
```

– empty() — jQuery-метод, который очищает содержимое элемента.

```
$('#parent').empty(); // Удаляет все дочерние элементы
```

Клонирование узлов в JavaScript — это процесс создания копии существующего элемента DOM. Это полезно, когда нужно размножить

элементы или перенести их в другие части страницы без потери оригинального содержимого.

– cloneNode(deep) — метод, который создаёт копию узла. Аргумент deep определяет, нужно ли клонировать только сам узел (false) или также его дочерние элементы (true);

```
let originalElement = document.getElementById('original');
let clonedElement = originalElement.cloneNode(true); // true для глубокого клонирования
document.body.appendChild(clonedElement);
```

– insertAdjacentHTML(position, html) — альтернатива, которая вставляет HTML-код рядом с существующим элементом. Хотя это не прямое клонирование, этот метод полезен для быстрого создания и вставки новых элементов.

```
let referenceElement = document.getElementById('reference');
referenceElement.insertAdjacentHTML('afterend', '<div>Новый элемент</div>');
```

Клонирование узлов позволяет эффективно управлять структурой DOM, создавая копии элементов и оптимизируя производительность веб-приложений.

События в JavaScript

События в JavaScript — это действия, которые происходят в браузере или вызванные действиями пользователя (таблица 16). Они позволяют создавать интерактивные веб-приложения, реагирующие на действия пользователя или изменения в системе.

Таблица 16 — События в JavaScript

Группа	**Событие**	**Описание**
события мыши	click	щелчок левой кнопкой мыши
	dblclick	двойной щелчок
	mouseover	наведение курсора на элемент
	mouseout	уход курсора с элемента
события клавиатуры	keydown	нажатие клавиши
	keyup	отпускание клавиши
	keypress	событие, возникающее при нажатии символа
события формы	submit	отправка формы
	change	изменение значения элемента формы
	focus	получение фокуса элементом
	blur	потеря фокуса элементом
события окна	resize	изменение размера окна

	scroll	прокрутка страницы
	load	загрузка страницы или ресурса
события таймера	setTimeout	выполнение функции через определённое время
	setInterval	периодическое выполнение функции

Есть несколько способов назначения обработчиков событий.

– Inline-обработчики (в HTML)

```
<button onclick="alert('Clicked!')">Click Me</button>
```

– Назначение через атрибуты JavaScript

```
let button = document.getElementById('myButton');
button.onclick = function() {
alert('Button clicked!');
};
```

– Использование метода addEventListener

```
let button = document.getElementById('myButton');
button.addEventListener('click', function() {
alert('Button clicked!');
});
```

Пример использования событий:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Event Example</title>
</head>
<body>
   <button id="myButton">Click Me</button>

   <script>
      let button = document.getElementById('myButton');
      button.addEventListener('click', function() {
         alert('Button was clicked!');
      });
   </script>
</body>
</html>
```

JavaScript и формы

Неотъемлемой частью веб-приложений являются формы, с помощью которых пользователи вводят данные и взаимодействуют с системой.

JavaScript предоставляет инструменты для работы с формами, начиная от базовой валидации и заканчивая отправкой данных на сервер.

В таблице 17 обобщены ключевые аспекты работы с формами в JavaScript для эффективного управления формой и её данными.

Таблица 17 — Работа с формами в JavaScript

Концепция	**Описание**	**Пример**
доступ к элементам формы	получение формы и её элементов через document.forms или document.getElementById	let form = document.getElementById('myForm');
валидация данных	проверка введённых данных перед отправкой формы	if (form.elements['username'].value === '') { alert('Username required!'); }
отправка формы	отправка формы с помощью метода submit()	form.submit();
предотвращение стандартной отправки	метод preventDefault() предотвращает стандартное поведение формы (перезагрузку страницы)	event.preventDefault();
работа с событиями формы	реакция на события формы, такие как submit, change, blur	form.addEventListener('submit', handleSubmit);
ajax и асинхронная отправка данных	использование XMLHttpRequest или fetch для отправки данных без перезагрузки страницы	fetch(url, { method: 'POST', body: JSON.stringify(formData) });
восстановление данных формы	использование reset() для очистки полей формы	form.reset();

Контрольные вопросы к разделу

1. Что такое HTML5 и какие основные задачи он решает в веб-разработке?
2. Из каких основных частей состоит структура HTML-документа?
3. Для чего используются теги <head> и <body>?
4. Что такое семантические элементы HTML и зачем они применяются?
5. Какие способы подключения CSS к HTML-документу существуют?
6. Что такое селекторы в CSS и какие базовые виды селекторов используются чаще всего?
7. В чём заключается принцип каскадности в CSS?
8. Что такое наследование стилей в CSS и как оно работает?
9. Какие основные свойства CSS используются для задания внешнего вида текста?

10. Как задаются цвета в CSS (основные способы записи)?
11. Что такое JavaScript и для чего он используется в веб-разработке?
12. Как JavaScript взаимодействует с HTML-страницей через DOM?
13. Что такое переменные в JavaScript и какие способы их объявления существуют?
14. Какие основные типы данных используются в JavaScript?
15. Что такое события в JavaScript и как они применяются для взаимодействия с пользователем?

ГЛАВА 3. СЕРВЕРНАЯ РАЗРАБОТКА

Серверная часть веб-приложения отвечает за обработку данных, реализацию бизнес-логики, взаимодействие с базой данных и интеграцию с внешними сервисами. Пользователь напрямую не взаимодействует с бэкендом, однако именно он обеспечивает корректную, безопасную и стабильную работу всех функций сайта или приложения. Современный backend выполняет значительно более широкий круг задач, чем простая передача HTML-страниц: он обрабатывает HTTP-запросы, управляет пользовательскими данными, обеспечивает авторизацию и аутентификацию, взаимодействует с базами данных, реализует REST API, интегрирует сторонние сервисы и AI-модели, контролирует безопасность системы, а также поддерживает масштабируемость и производительность веб-приложения.

Архитектура бэкенда

Типичная архитектура серверной части строится на взаимодействии основных компонентов:

- Сервер — программное обеспечение, которое постоянно запущено и ожидает входящие запросы. Он анализирует их, выполняет необходимые вычисления и отправляет ответ обратно клиенту.
- База данных (БД) — специализированная система для хранения и управления данными. БД обеспечивает надёжное сохранение информации, быстрый поиск и целостность данных.
- API (Application Programming Interface) — программный интерфейс, который определяет, как клиентское приложение может взаимодействовать с сервером. Через API фронтенд запрашивает данные или отправляет команды на выполнение.
- Бизнес-логика — ядро приложения, где реализована вся его функциональность. Это набор кода, который обрабатывает данные согласно правилам, заложенным разработчиками.

В таблице 18 представлены языки программирования и популярные фреймворки, используемые для серверной разработки.

Таблица 18 — Языки программирования и фреймворки для серверной разработки

Язык программирования	**Популярные фреймворки**	**Особенности и применение**
Python	Django, Flask, FastAPI	Высокая скорость разработки, читаемость кода, лидер в области Data Science и ИИ.
JavaScript (Node.js)	Express.js, NestJS	Единый язык для фронта и бэка, высокая производительность при работе с I/O-операциями (чаты, стриминг).
Java	Spring Boot	Стандарт для крупных корпоративных систем (Enterprise), высокая надежность и производительность.
PHP	Laravel, Symfony	Исторически самый популярный язык для веба, огромная экосистема и простота хостинга.
Go (Golang)	Gin, Echo	Высокая скорость компиляции и выполнения, эффективная работа с многопоточностью (конкурентностью).
C#	.NET (ASP.NET Core)	Платформа от Microsoft, идеальна для интеграции с инфраструктурой Windows и корпоративных решений.

Взаимодействие с фронтендом: REST API

API (Application Programming Interface) представляет собой набор правил и инструментов, который позволяет одной компьютерной программе взаимодействовать с другой. REST (Representational State Transfer) — это архитектурный стиль, набор рекомендаций для построения таких интерфейсов. Он не является стандартом, а скорее сводом лучших практик, которые делают взаимодействие предсказуемым и простым.

Ключевая идея REST: всё есть ресурс. Ресурсом может быть что угодно: пользователь, товар в магазине, статья в блоге, комментарий. Каждый ресурс имеет свой уникальный идентификатор — URL (Uniform Resource Locator).

В REST для работы с ресурсами используются стандартные методы протокола HTTP (таблица 19). Каждый метод имеет четкое назначение, что делает код понятным.

Таблица 19 — Основные HTTP-методы

Метод	**Действие**	**Пример (для ресурса "статьи")**
GET	Прочитать (получить) данные. Запросы безопасны и не изменяют состояние сервера	`GET /api/articles` — получить список всех статей
		`GET /api/articles/144` — получить статью с ID 144
POST	Создать новый ресурс	`POST /api/articles` — отправить данные для создания новой статьи.
PUT	Обновить существующий ресурс целиком	`PUT /api/articles/144` — полностью заменить статью с ID 144 на новую
PATCH	Частично обновить ресурс	`PATCH /api/articles/144` — изменить только заголовок статьи с ID 144
DELETE	Удалить ресурс	`DELETE /api/articles/144` — удалить статью с ID 144

Форматы данных для API

Когда фронтенд-приложение и бэкенд обмениваются информацией, им необходим общий формат сериализации данных. Он определяет, как структурировать и упаковывать информацию, чтобы обе стороны могли её корректно прочитать и обработать. Выбор формата данных влияет на скорость работы приложения, объём передаваемого трафика и удобство разработки.

JSON (JavaScript Object Notation). JSON широко используется для обмена данными в современных API. Это текстовый формат обмена данными, основанный на подмножестве синтаксиса JavaScript. Данные в JSON представлены в виде коллекций пар «ключ-значение» и упорядоченных списков значений.

```
{
"user": {
   "id": 101,
   "name": "Иван",
   "email": "ivan@example.com",
   "is_active": true,
   "roles": ["user", "moderator"],
```

```
    "profile": null
  }
{
```

Формат JSON получил широкое распространение в веб-разработке благодаря своей компактности, простоте и удобству использования. Одним из его главных преимуществ является легковесность: JSON занимает меньше места по сравнению с XML, который долгое время использовался для обмена данными между системами. Благодаря этому уменьшается объем передаваемого трафика и повышается скорость взаимодействия между клиентом и сервером.

Еще одним важным достоинством JSON является его высокая читаемость. Структура данных в формате JSON интуитивно понятна не только компьютеру, но и человеку, что упрощает разработку, тестирование и отладку приложений. Кроме того, практически все современные языки программирования содержат встроенные библиотеки для работы с JSON, а в JavaScript данный формат фактически является нативным типом представления данных. Это делает JSON универсальным инструментом обмена информацией в веб-приложениях.

JSON также поддерживает иерархическую структуру данных, позволяя описывать сложные объекты с вложенными элементами, массивами и связями между ними. Благодаря этому формат широко применяется при передаче данных в REST API, мобильных приложениях, облачных сервисах и системах искусственного интеллекта.

Несмотря на многочисленные преимущества, JSON имеет и некоторые недостатки. Одним из основных является отсутствие строгой схемы описания структуры данных. По умолчанию JSON не содержит встроенного механизма проверки формата передаваемой информации, поэтому изменения структуры ответа на сервере могут привести к ошибкам на стороне клиента. Для решения данной проблемы используются дополнительные инструменты, например JSON Schema, позволяющие задавать правила валидации и контроля структуры данных.

XML (eXtensible Markup Language). Формат XML долгое время являлся основным стандартом обмена данными между информационными системами и до сих пор широко используется в корпоративной среде. Одним из его главных преимуществ является строгость структуры и высокая самодокументируемость. Данные заключены в теги, которые можно произвольно именовать, создавая иерархическую структуру, похожую на HTML. Теги XML позволяют четко описывать содержимое данных, благодаря чему структура документа становится понятной как разработчикам, так и программным системам.

```
<user id="101">
  <name>Иван</name>
  <email>ivan@example.com</email>
  <is_active>true</is_active>
  <roles>
    <role>user</role>
    <role>moderator</role>
  </roles>
</user>
```

Важной особенностью XML является поддержка схем XSD (XML Schema Definition), которые позволяют строго задавать структуру документа, определять типы данных, обязательные поля и правила валидации. Это делает XML особенно востребованным в банковских системах, государственных сервисах и крупных корпоративных приложениях, где требуется высокий уровень надежности и контроля данных. Кроме того, XML поддерживается большим количеством программных платформ и интеграционных решений, что обеспечивает его совместимость с различными информационными системами.

Однако XML имеет и ряд недостатков, основным из которых является избыточность структуры. Большое количество тегов значительно увеличивает размер файлов и объем передаваемых данных. Это может снижать производительность веб-приложений и увеличивать сетевой трафик. Кроме

того, обработка XML требует больше вычислительных ресурсов, поскольку его парсинг сложнее по сравнению с JSON. По этой причине в современных веб-приложениях и REST API чаще используется JSON, тогда как XML преимущественно сохраняет позиции в корпоративных системах и специализированных областях.

YAML (YAML Ain't Markup Language). Формат, ориентированный на максимальную читаемость человеком. Он часто используется для файлов конфигурации (docker-compose, Kubernetes), но редко применяется непосредственно в API-запросах/ответах из-за сложности парсинга и неоднозначности типов данных. Использует отступы (пробелы) для определения иерархии и - для обозначения элементов списка.

```
user:
  id: 101
  name: Иван
  email: ivan@example.com
  is_active: true
  roles:
    - user
    - moderator
```

YAML является современным форматом представления данных, который широко используется для настройки приложений, описания инфраструктуры и работы с системами автоматизации. Его главным преимуществом считается высокая читаемость и простота синтаксиса. YAML использует минимальное количество специальных символов, а структура документа задается с помощью отступов, благодаря чему файлы выглядят понятными и близкими к обычному тексту. По сравнению с JSON и XML такой подход уменьшает «визуальный шум» и облегчает восприятие информации.

Еще одним преимуществом YAML является компактность. Для описания одинаковых данных обычно требуется меньше строк, чем в XML или JSON,

поскольку отсутствуют закрывающие теги и большое количество служебных символов. YAML также поддерживает сложные структуры данных: вложенные объекты, списки, ссылки и повторное использование элементов. Кроме того, формат поддерживает различные типы данных, включая логические значения, многострочные строки и комментарии, что особенно удобно при создании конфигурационных файлов. Как и JSON, YAML является языконезависимым форматом и поддерживается большинством современных языков программирования.

Несмотря на удобство, YAML имеет ряд недостатков. Основной проблемой является высокая чувствительность к отступам: даже ошибка в одном пробеле может нарушить структуру документа и привести к сбоям при обработке данных. Кроме того, YAML иногда неоднозначно интерпретирует типы данных, автоматически определяя логические значения или числа, что может вызывать ошибки. Парсинг YAML значительно сложнее по сравнению с JSON, поэтому обработка таких файлов требует больше ресурсов и может снижать производительность системы. По этой причине YAML чаще применяется для конфигурации и настройки приложений, тогда как JSON остается более популярным форматом передачи данных в веб-разработке и REST API.

В некоторых специализированных областях веб-разработки и распределенных систем применяются и другие форматы передачи данных. Одним из таких форматов является Protobuf (Protocol Buffers), разработанный компанией Google. В отличие от JSON и XML, Protobuf представляет данные в бинарном виде, поэтому он не предназначен для чтения человеком. Основными преимуществами данного формата являются высокая скорость сериализации и десериализации данных, а также минимальный размер передаваемых сообщений. Благодаря этим особенностям Protobuf широко используется в высоконагруженных системах, микросервисной архитектуре и технологии gRPC.

Еще одним примером является BSON — бинарное представление JSON, применяемое в СУБД MongoDB. BSON расширяет возможности JSON, добавляя поддержку дополнительных типов данных и повышая эффективность хранения информации в базе данных.

Для большинства современных веб-приложений и REST API наиболее распространенным и универсальным решением остается JSON. Этот формат обеспечивает оптимальный баланс между читаемостью, простотой разработки и эффективностью передачи данных. Другие форматы обычно используются только при наличии специальных требований, например необходимости строгой валидации структуры данных в XML или достижения максимальной производительности при использовании Protobuf.

Базы данных

Любой API, кроме самых простых, работает с данными. Любое современное веб-приложение, от простого блога до сложной социальной сети, работает с данными. Пользователи, их посты, комментарии, товары в корзине — вся эта информация должна где-то храниться, быстро извлекаться и надежно защищаться. Эту задачу выполняют базы данных.

База данных (БД) — это организованная коллекция данных, хранящаяся в электронном виде. Для работы с БД используются системы управления базами данных, программное обеспечение, которое позволяет создавать, изменять и управлять данными. Именно СУБД гарантирует, что данные не противоречат друг другу и не повреждаются, позволяет настраивать права доступа, обеспечивает быстрый поиск, и позволяет осуществлять параллельный доступ множеству пользователей одновременно.

В веб-разработке выделяют два основных семейства баз данных: реляционные (SQL) и нереляционные (NoSQL). Выбор между ними является одним из важнейших архитектурных решений при проектировании веб-приложения, поскольку от типа базы данных зависят производительность системы, способы хранения информации и возможности масштабирования.

Реляционные системы управления базами данных представляют собой классический подход к хранению информации. В таких базах данные организуются в виде таблиц, состоящих из строк и столбцов, а таблицы могут быть связаны между собой с помощью ключей. Для работы с реляционными базами данных используется язык SQL (Structured Query Language). Наиболее распространенными примерами являются PostgreSQL, MySQL, Microsoft SQL Server и SQLite.

Главным преимуществом реляционных СУБД является строгая структура данных. Разработчик заранее определяет схему базы данных, типы данных и связи между таблицами, что позволяет избежать ошибок и поддерживать целостность информации. Кроме того, SQL-базы поддерживают транзакции и принципы ACID, благодаря которым критически важные операции выполняются надежно и последовательно. Реляционные базы данных особенно удобны для реализации сложных связей между сущностями, например, отношений «один ко многим» или «многие ко многим». Поэтому такие СУБД широко применяются в интернет-магазинах, банковских системах, бухгалтерских приложениях и других проектах, где особенно важны точность и согласованность данных.

Нереляционные базы данных появились как более гибкая альтернатива традиционным SQL-системам. Термин NoSQL расшифровывается как «Not Only SQL» — «не только SQL». Такие базы данных не требуют строгой табличной структуры и позволяют хранить данные в различных форматах. Существует несколько типов NoSQL-баз данных. Документоориентированные СУБД, такие как MongoDB и Couchbase, хранят данные в виде документов, похожих на JSON. Системы типа «ключ-значение», например, Redis и Memcached, обеспечивают очень высокую скорость работы и часто используются для кэширования. Колоночные базы данных, такие как Apache Cassandra, предназначены для обработки огромных объемов информации, а графовые СУБД, например, Neo4j, эффективно работают со сложными связями между объектами.

Основным преимуществом NoSQL-систем является гибкость. Разработчик может изменять структуру данных без полной перестройки базы данных, добавляя новые поля и типы информации в процессе работы приложения. Кроме того, NoSQL-базы легче масштабируются горизонтально, то есть нагрузка может распределяться между несколькими серверами. Такие системы хорошо подходят для обработки больших потоков данных и высоконагруженных приложений. Поэтому NoSQL часто используется в социальных сетях, системах аналитики в реальном времени, сервисах хранения логов, рекомендательных системах и механизмах кэширования.

Взаимодействие бэкенда с базой данных

В современных веб-приложениях взаимодействие с базой данных обычно не выполняется напрямую. Между программным кодом и системой управления базами данных используется специальный промежуточный слой — драйвер СУБД или ORM (Object-Relational Mapping). Такой подход упрощает разработку, делает код более понятным и повышает безопасность работы с данными.

Драйвер СУБД представляет собой низкоуровневую библиотеку, которая обеспечивает соединение приложения с сервером базы данных. С помощью драйвера программа на языке Python, JavaScript или другом языке может отправлять SQL-запросы, получать результаты и обрабатывать их. Например, драйвер позволяет выполнить запрос к базе данных, получить список пользователей или сохранить новую запись. Работа через драйвер требует знания SQL и ручного написания запросов, однако обеспечивает высокий уровень контроля над взаимодействием с базой данных.

Для упрощения разработки во многих современных фреймворках используется ORM, который является надстройкой над драйвером СУБД и позволяет работать с базой данных в объектно-ориентированном стиле. В этом случае таблицы базы данных представляются в виде классов, а записи — в виде

объектов языка программирования. Такой подход делает код более удобным, читаемым и близким к логике приложения.

Например, вместо написания SQL-запроса:

sql

```
SELECT * FROM users WHERE id = 1;
```

разработчик может использовать объектно-ориентированный код:

python

```
user = User.query.get(1)
```

В данном случае ORM автоматически преобразует команду в SQL-запрос, отправит его в базу данных, получит результат и преобразует его обратно в объект `user`, с которым можно работать в программе. Разработчику не требуется вручную писать SQL для большинства типовых операций.

Использование ORM имеет ряд преимуществ. Во-первых, значительно ускоряется разработка приложений, так как уменьшается количество однотипного кода. Во-вторых, ORM помогает снизить вероятность ошибок и повышает безопасность системы, автоматически защищая приложение от некоторых видов атак, например, SQL-инъекций. Кроме того, ORM облегчает перенос приложения между различными СУБД, поскольку разработчик работает не с конкретным SQL-диалектом, а с универсальными объектами и методами.

Однако ORM не является универсальным решением для всех задач. При работе со сложными аналитическими запросами или высоконагруженными системами ручное написание SQL может быть более эффективным и производительным. Поэтому на практике разработчики часто совмещают ORM и прямые SQL-запросы в зависимости от требований проекта.

В экосистеме Python одним из наиболее популярных ORM является SQLAlchemy, широко применяемый во фреймворке Flask. Для фреймворка Django используется встроенный Django ORM, а в среде JavaScript популярностью пользуются Sequelize и Prisma. Благодаря ORM работа с базами

данных становится более удобной и доступной даже для начинающих разработчиков.

Взаимодействие фронтенда с базой данных

В современной веб-разработке фронтенд-приложение никогда не должно обращаться к базе данных напрямую. Правильная архитектура предполагает, что взаимодействие клиента с данными всегда происходит через серверную часть приложения — backend. Прямое подключение фронтенда к базе данных считается плохой практикой и представляет серьезную угрозу безопасности. Базы данных содержат конфиденциальную информацию: пароли пользователей, персональные данные, финансовую информацию и другие критически важные сведения. Если предоставить клиентскому приложению прямой доступ к базе данных, злоумышленник сможет с помощью инструментов разработчика браузера отправлять произвольные запросы, получать закрытые данные, изменять записи или даже удалять информацию из базы данных. Именно поэтому база данных должна быть изолирована от прямого доступа пользователей.

Еще одной важной причиной использования backend является необходимость реализации бизнес-логики приложения. База данных сама по себе является лишь системой хранения информации и не знает правил работы приложения. Например, она не может определить, что скидка предоставляется только новым пользователям или действует лишь в определенный день недели. Проверка прав доступа, расчет скидок, валидация данных, обработка платежей и другие правила работы системы должны выполняться именно серверной частью приложения.

Кроме того, прямое обращение к базе данных с клиента негативно сказалось бы на производительности и усложнило бы разработку. Фронтенд-приложения часто работают на мобильных устройствах и в условиях нестабильного интернет-соединения. Формирование сложных SQL-запросов на

стороне клиента создало бы повышенную нагрузку на систему и усложнило обеспечение безопасности данных.

Правильная схема взаимодействия выглядит следующим образом: фронтенд отправляет запрос к backend API, например, запрос на получение профиля пользователя. Сервер принимает этот запрос и выполняет несколько этапов обработки. Сначала backend проверяет, авторизован ли пользователь и имеет ли он право доступа к данным. Затем сервер применяет бизнес-логику приложения, например, проверяет, активен ли аккаунт пользователя или не заблокирован ли профиль. После этого backend формирует безопасный SQL-запрос к базе данных, получает необходимые данные, обрабатывает их и отправляет клиенту только ту информацию, которая действительно нужна. Например, сервер может скрыть пароль, служебные поля или конфиденциальные данные перед отправкой ответа на frontend.

Коды состояния HTTP

При взаимодействии клиента и сервера в веб-приложениях сервер отправляет не только данные, но и специальный код состояния HTTP (HTTP Status Code), который сообщает результат обработки запроса. Эти коды представляют собой трехзначные числа и являются важной частью протокола HTTP. Благодаря кодам состояния клиентское приложение, браузер или API могут понять, успешно ли выполнен запрос, произошла ли ошибка со стороны пользователя или проблема возникла на сервере.

Коды состояния делятся на несколько основных групп. Наиболее часто в веб-разработке используются группы 2xx, 4xx и 5xx. Первая цифра кода определяет общий тип ответа сервера.

Коды группы 2xx обозначают успешное выполнение запроса. Например, код 200 OK означает, что запрос был успешно обработан и сервер вернул запрашиваемые данные. Такой код чаще всего используется при выполнении GET-запросов. Код 201 Created сообщает, что новый ресурс был успешно создан на сервере. Обычно этот код возвращается после выполнения POST-

запросов, например, при регистрации пользователя или создании новой записи в базе данных.

Коды группы 4xx указывают на ошибки со стороны клиента. Это означает, что проблема связана с самим запросом пользователя. Код 400 Bad Request означает неверный синтаксис запроса или отсутствие необходимых данных. Код 401 Unauthorized возвращается в случае, если пользователь не прошел аутентификацию и должен выполнить вход в систему. Код 403 Forbidden означает, что пользователь авторизован, однако не имеет прав доступа к запрашиваемому ресурсу. Одним из наиболее известных кодов является 404 Not Found, который сообщает о том, что запрашиваемая страница или ресурс не существуют на сервере.

Коды группы 5xx обозначают ошибки на стороне сервера. В таких случаях проблема возникает не из-за пользователя, а из-за внутреннего сбоя приложения или сервера. Наиболее распространенным является код 500 Internal Server Error, который указывает на непредвиденную ошибку в работе backend-приложения. Причиной могут быть ошибки программного кода, проблемы с базой данных или сбои серверной инфраструктуры.

Использование HTTP-кодов состояния является важной частью разработки REST API и современных веб-приложений. Они помогают frontend-приложениям корректно реагировать на ответы сервера, отображать сообщения об ошибках пользователю и автоматически обрабатывать различные сценарии взаимодействия. Правильное использование кодов состояния также упрощает тестирование, отладку и сопровождение веб-систем.

Контейнеризация в веб-разработке

В современной веб-разработке одной из наиболее распространенных проблем является различие окружений разработки и эксплуатации приложения. Нередко приложение корректно работает на компьютере разработчика, но при переносе на тестовый или production-сервер возникают ошибки, связанные с

версиями операционной системы, библиотек, интерпретаторов языков программирования или настройками сервера. Именно поэтому среди разработчиков получила широкую известность фраза: «У меня на локальной машине всё работало».

Для решения данной проблемы используется технология контейнеризации, одной из самых популярных платформ которой является Docker. Docker позволяет упаковать приложение вместе со всеми его зависимостями в специальный изолированный контейнер. Такой контейнер будет одинаково работать на любом компьютере, сервере или облачной платформе независимо от особенностей операционной системы и окружения.

Контейнеризация стала важной частью современной backend-разработки, DevOps-подхода и облачных технологий. Docker широко применяется при разработке веб-приложений, REST API, микросервисов, AI-сервисов и систем машинного обучения.

Основные понятия Docker

Для понимания принципов работы Docker необходимо различать два ключевых понятия: образ (Image) и контейнер (Container).

Docker-образ представляет собой шаблон или «чертеж» приложения. Это специальный файл, содержащий все необходимое для запуска программы: исходный код приложения, среду выполнения (например, Python или Node.js), библиотеки, зависимости, системные инструменты и переменные окружения. Образ является неизменяемым и создается на основе инструкций, записанных в файле Dockerfile.

Docker-контейнер — это уже запущенный экземпляр образа. Контейнер представляет собой изолированную среду выполнения, внутри которой работает приложение. Из одного образа можно создать множество контейнеров, каждый из которых будет функционировать независимо от других.

Для понимания различия можно использовать следующую аналогию: Docker-образ похож на установочный ISO-образ операционной системы, а

контейнер — на уже запущенную виртуальную машину, созданную на его основе.

Одним из главных преимуществ Docker является обеспечение однородности окружения. Приложение, работающее в Docker-контейнере на компьютере разработчика, будет работать аналогичным образом на тестовом сервере, в облаке или production-среде. Это существенно уменьшает количество ошибок, связанных с несовместимостью программного окружения.

Еще одним важным преимуществом является изоляция приложений. Каждый сервис запускается в собственном контейнере и использует свои зависимости, библиотеки и настройки. Благодаря этому на одном сервере можно одновременно запускать несколько проектов с разными версиями Python, Node.js или других технологий без возникновения конфликтов.

Docker также значительно упрощает развертывание приложений. Вместо длительной ручной настройки сервера достаточно запустить контейнер с помощью одной команды. Это ускоряет процесс публикации веб-приложений и делает инфраструктуру более предсказуемой и управляемой.

Технология контейнеризации особенно важна при использовании микросервисной архитектуры. В современных системах большое приложение часто разделяется на набор независимых сервисов: backend API, база данных, frontend, сервис авторизации, AI-модуль и другие компоненты. Каждый такой сервис может работать в отдельном Docker-контейнере, что облегчает разработку, тестирование, обновление и масштабирование системы.

Сегодня Docker активно используется совместно с облачными платформами и инструментами автоматизации. Для управления большим количеством контейнеров применяется система оркестрации Kubernetes. В экосистеме веб-разработки Docker особенно популярен при создании приложений на Flask, Django, Node.js и других backend-фреймворках. Контейнеризация позволяет разработчикам быстро создавать воспроизводимое окружение, упрощает командную работу и делает процесс развертывания веб-приложений более надежным и автоматизированным.

Контрольные вопросы к разделу

1. Что такое бэкенд и какие задачи он выполняет в веб-приложении?
2. Какие основные компоненты включает архитектура серверной части приложения?
3. В чём заключается взаимодействие между фронтендом и бэкендом?
4. Что такое REST API и какие принципы лежат в его основе?
5. Какие HTTP-методы чаще всего используются в REST API и для чего они предназначены?
6. Какие форматы данных применяются для обмена информацией между клиентом и сервером?
7. В чём преимущества формата JSON по сравнению с другими форматами данных?
8. Что такое база данных и какие основные типы баз данных используются в веб-разработке?
9. В чём различие между реляционными и нереляционными базами данных?
10. Как происходит взаимодействие бэкенда с базой данных?
11. Может ли фронтенд напрямую взаимодействовать с базой данных? Объясните почему.
12. Что такое HTTP-коды состояния и зачем они используются в веб-приложениях? Какие группы HTTP-кодов состояния существуют и что они обозначают?
13. Что такое контейнеризация в веб-разработке и какие преимущества она предоставляет?

ГЛАВА 4. ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ В ВЕБ-РАЗРАБОТКЕ

Искусственный интеллект (Artificial Intelligence, AI) представляет собой направление информатики, связанное с созданием систем, способных выполнять задачи, требующие интеллектуальной деятельности человека. К таким задачам относятся анализ данных, распознавание изображений и речи, обработка текста, прогнозирование и принятие решений.

Технологии искусственного интеллекта активно внедряются в современные веб-приложения и онлайн-сервисы. Их использование повышает удобство работы пользователей, улучшает качество обслуживания и позволяет автоматизировать бизнес-процессы. Искусственный интеллект помогает веб-приложениям быстрее обрабатывать информацию, адаптироваться под потребности пользователей и выполнять сложные задачи без постоянного участия человека.

Одним из наиболее распространённых направлений применения ИИ являются интеллектуальные чат-боты. Чат-боты способны автоматически отвечать на вопросы пользователей, консультировать клиентов и помогать в навигации по сайту. Современные боты используют технологии обработки естественного языка (Natural Language Processing, NLP), благодаря которым система может понимать смысл пользовательских запросов и формировать ответы, максимально приближенные к человеческой речи. Большие языковые модели позволяют создавать виртуальных помощников, поддерживающих диалог, анализирующих контекст общения и способных решать широкий круг задач — от технической поддержки до оформления заказов.

Важную роль в современных веб-сервисах играют рекомендательные системы. Интернет-магазины, стриминговые платформы и социальные сети используют алгоритмы искусственного интеллекта для анализа предпочтений пользователей и формирования персональных рекомендаций. Такие системы изучают историю просмотров, покупок, оценок и действий пользователя, после

чего предлагают наиболее подходящие товары, фильмы, музыку или публикации. Использование рекомендательных алгоритмов позволяет повысить удобство работы с сервисом и увеличить заинтересованность пользователей.

Искусственный интеллект широко применяется для обработки изображений. Современные алгоритмы способны распознавать лица, анализировать фотографии, определять объекты на изображениях и автоматически модерировать контент. Например, системы компьютерного зрения используются в социальных сетях для распознавания людей на фотографиях, а также в интернет-магазинах для поиска похожих товаров по изображению. Технологии обработки изображений помогают автоматизировать задачи, которые ранее требовали участия человека.

Одним из наиболее быстро развивающихся направлений является генерация контента. Системы генеративного искусственного интеллекта способны автоматически создавать тексты, описания товаров, изображения, элементы интерфейса и даже программный код. Веб-разработчики используют генеративные модели для ускорения создания сайтов, подготовки контента и проектирования дизайна. Такие технологии позволяют значительно сократить время разработки и автоматизировать рутинные процессы.

Алгоритмы машинного обучения активно используются в аналитике и прогнозировании. Искусственный интеллект помогает анализировать поведение пользователей, выявлять закономерности и прогнозировать спрос на товары и услуги. Кроме того, ИИ способен обнаруживать аномалии в работе веб-приложений, выявлять подозрительную активность и помогать в обеспечении информационной безопасности. Аналитические системы на основе машинного обучения позволяют компаниям принимать более точные управленческие решения на основе больших объёмов данных.

Современные веб-приложения всё чаще используют голосовые интерфейсы. Технологии распознавания речи позволяют пользователям взаимодействовать с сайтами и сервисами с помощью голосовых команд. Системы синтеза речи, в свою очередь, обеспечивают озвучивание текстовой

информации. Голосовые интерфейсы повышают доступность веб-приложений для различных категорий пользователей и делают взаимодействие с сервисами более удобным.

Интеграция искусственного интеллекта в веб-приложения обычно выполняется через внешние API и облачные сервисы. Такой подход позволяет разработчикам использовать мощные модели машинного обучения без необходимости самостоятельно обучать нейронные сети и поддерживать сложную вычислительную инфраструктуру. Облачные ИИ-сервисы предоставляют готовые инструменты для обработки текста, изображений, аудио и других данных, что значительно упрощает внедрение интеллектуальных функций в современные веб-приложения.

Применение искусственного интеллекта в веб-приложениях

Использование искусственного интеллекта позволяет автоматизировать взаимодействие с пользователями, улучшать качество сервисов, повышать уровень персонализации и существенно ускорять разработку цифровых продуктов. Благодаря ИИ веб-приложения становятся более «умными», адаптивными и ориентированными на потребности конкретного пользователя. Рассмотрим основные направления применения искусственного интеллекта в веб-разработке.

Чат-боты и виртуальные ассистенты

Чат-боты и виртуальные ассистенты решают задачу автоматизации общения с пользователями. Они применяются для консультирования клиентов, обработки входящих запросов, помощи в навигации по сайту, оформления заказов и предоставления технической поддержки в круглосуточном режиме.

Например, в интернет-магазине чат-бот может помочь пользователю подобрать товар по заданным параметрам, проверить его наличие на складе и оформить доставку без участия оператора. Это снижает нагрузку на службу поддержки и ускоряет обслуживание клиентов.

Современные чат-боты основаны на технологиях обработки естественного языка (Natural Language Processing, NLP) и больших языковых моделях (Large Language Models, LLM). Такие системы анализируют текст запроса пользователя, определяют его смысл и контекст, после чего формируют наиболее подходящий ответ. Дополнительно применяются методы машинного обучения, которые позволяют улучшать качество ответов на основе накопленного опыта взаимодействия с пользователями.

Рекомендательные системы

Рекомендательные системы предназначены для подбора персонализированного контента или товаров для пользователя. Их основная цель — повысить удобство использования сервиса и увеличить вовлечённость пользователя за счёт индивидуальных предложений.

На практике такие системы используются в различных цифровых сервисах:

- рекомендации фильмов и сериалов в стриминговых платформах;
- подбор товаров в интернет-магазинах;
- предложения друзей, групп или контента в социальных сетях.

Принцип работы рекомендательных систем основан на анализе поведения пользователя и сравнении его с поведением других пользователей, имеющих схожие интересы.

Для формирования персонализированных рекомендаций используются различные типы данных о пользователе и его действиях. К ним относятся:

- история просмотров и покупок;
- оценки, лайки и дизлайки;
- поисковые запросы;
- время взаимодействия с контентом;
- данные профиля пользователя (возраст, интересы, регион и др.).

Чем больше данных доступно системе, тем точнее и релевантнее становятся рекомендации, формируемые алгоритмами искусственного интеллекта.

Подходы к построению рекомендательных систем

Существует несколько основных подходов к построению рекомендательных систем.

Коллаборативная фильтрация - основана на анализе поведения пользователей с похожими предпочтениями. Если пользователи с похожими действиями выбирают определённый контент, система может рекомендовать его другим похожим пользователям.

Контентный подход - использует анализ характеристик самих объектов, например жанра фильма, категории товара или ключевых слов в описании.

Гибридные системы - объединяют несколько методов одновременно, что позволяет повысить точность и устойчивость рекомендаций.

Например, потоковый сервис может рекомендовать фильм, учитывая как предпочтения пользователя, так и популярность фильма среди похожей аудитории.

Аналитика поведения пользователей

Искусственный интеллект активно применяется для анализа поведения пользователей в веб-приложениях. Системы собирают и обрабатывают данные о взаимодействии пользователя с интерфейсом.

К таким данным относятся:

- клики и переходы по страницам;
- время нахождения на сайте;
- последовательность действий;
- взаимодействие с элементами интерфейса.

На основе анализа этих данных можно:

- улучшать структуру и дизайн интерфейса;
- повышать конверсию веб-приложений;

- выявлять проблемные места в пользовательском опыте;
- прогнозировать дальнейшее поведение пользователей.

Например, если система фиксирует, что пользователи часто покидают страницу оформления заказа, она может предложить упростить форму оплаты или изменить интерфейс.

Обработка изображений и текста

Искусственный интеллект решает широкий спектр задач в области обработки изображений и текстовой информации.

Обработка изображений включает:

- распознавание объектов и лиц;
- автоматическую модерацию контента;
- поиск изображений по содержимому;
- улучшение качества изображений.

Обработка текста включает:

- автоматический перевод;
- анализ тональности отзывов;
- классификацию текстов;
- генерацию текстового контента.

Например, социальные сети используют ИИ для автоматического выявления нежелательного контента и соблюдения правил платформы.

Технологии распознавания и генерации текста

Основой современных систем обработки текста являются технологии обработки естественного языка, методы машинного обучения и глубокого обучения.

Ключевыми технологиями являются:

- модели машинного обучения;
- архитектура трансформеров;
- большие языковые модели.

Эти технологии позволяют системам анализировать смысл текста, определять его тональность, классифицировать сообщения и генерировать новые тексты.

Например, система может автоматически определить, является ли отзыв пользователя положительным или отрицательным, а также сформировать краткое резюме большого текста.

Голосовые интерфейсы

Голосовые интерфейсы позволяют пользователям взаимодействовать с веб-приложениями с помощью речи. Это делает взаимодействие более естественным и удобным, особенно в мобильных и интеллектуальных системах.

Основные технологии включают:

- распознавание речи (Speech-to-Text);
- синтез речи (Text-to-Speech);
- обработку естественного языка.

Голосовые интерфейсы применяются в виртуальных помощниках, системах поиска, навигации и управления устройствами.

Например, пользователь может произнести команду: «Найди ближайшую кофейню», и система автоматически выполнит поиск и предоставит результаты.

Автоматизация веб-разработки с помощью ИИ

Искусственный интеллект значительно ускоряет и упрощает процесс веб-разработки. Он используется на всех этапах создания веб-приложений — от написания кода до тестирования и документирования.

Основные направления применения:

- генерация HTML, CSS и JavaScript-кода;
- автодополнение и подсказки в редакторах кода;
- поиск и исправление ошибок;
- автоматическое создание тестов;

- генерация технической документации.

Например, разработчик может описать интерфейс словами, а система искусственного интеллекта сгенерирует готовую HTML-страницу с применёнными стилями CSS и базовой логикой на JavaScript.

Также ИИ помогает анализировать код, выявлять потенциальные ошибки и предлагать оптимизации, что повышает качество программного обеспечения и ускоряет разработку.

Таким образом, искусственный интеллект охватывает практически все ключевые аспекты работы современных веб-приложений — от взаимодействия с пользователем и анализа данных до автоматизации разработки и создания интеллектуальных сервисов.

Этические аспекты использования искусственного интеллекта

Развитие технологий искусственного интеллекта сопровождается рядом этических и правовых вопросов. Современные ИИ-системы активно используются в веб-приложениях, онлайн-сервисах, системах анализа данных и цифровых платформах, поэтому важно учитывать возможные риски и последствия их применения. Ответственное использование искусственного интеллекта становится важной задачей как для разработчиков программного обеспечения, так и для организаций, внедряющих интеллектуальные технологии.

Одной из наиболее значимых проблем является конфиденциальность данных. Для обучения моделей машинного обучения и работы интеллектуальных сервисов часто используются большие объёмы информации о пользователях, включая персональные данные. В связи с этим разработчики обязаны обеспечивать надёжную защиту информации, предотвращать утечки данных и соблюдать требования законодательства в области информационной безопасности и защиты персональных данных. Нарушение

конфиденциальности может привести не только к техническим и финансовым потерям, но и к утрате доверия со стороны пользователей.

Другой важной задачей является предотвращение дискриминации и предвзятости алгоритмов. Системы искусственного интеллекта обучаются на данных, предоставленных человеком, поэтому качество и объективность обучающих наборов напрямую влияют на результаты работы модели. Если данные содержат ошибки, неточности или социальные стереотипы, алгоритм может воспроизводить несправедливые решения и формировать предвзятые выводы. Например, интеллектуальная система может некорректно оценивать пользователей или выдавать необъективные рекомендации. По этой причине разработчики должны тщательно анализировать данные и контролировать корректность работы моделей.

Особое внимание необходимо уделять достоверности генерируемого контента. Большие языковые модели способны создавать тексты, внешне похожие на материалы, написанные человеком, однако информация в таких текстах может содержать ошибки или недостоверные сведения. Иногда искусственный интеллект формирует убедительные, но фактически неверные ответы. Поэтому результаты работы ИИ требуют обязательной проверки человеком, особенно в сферах, связанных с образованием, медициной, финансами и правовыми вопросами.

Важным аспектом является прозрачность работы алгоритмов искусственного интеллекта. Пользователи должны понимать, каким образом система принимает решения и какие данные используются для её работы. Прозрачность помогает повысить доверие к интеллектуальным сервисам и облегчает выявление возможных ошибок в работе алгоритмов.

Не менее важным принципом является ответственность разработчиков и организаций, внедряющих ИИ-технологии. Специалисты должны учитывать возможные последствия использования интеллектуальных систем, обеспечивать безопасность программных решений и предотвращать злоупотребление технологиями. Ответственный подход к разработке позволяет

снизить риски и обеспечить более безопасное использование искусственного интеллекта.

Ответственное использование искусственного интеллекта является важной частью современной профессиональной деятельности веб-разработчика. Специалист, работающий с ИИ-технологиями, должен учитывать не только технические возможности систем, но и их влияние на людей, общество и информационную безопасность.

Контрольные вопросы к разделу

1. Какие преимущества дает внедрение ИИ в веб-приложения?
2. Какие проблемы могут возникнуть при использовании ИИ?
3. Почему важно соблюдать этические нормы при разработке ИИ-систем?
4. Какие основные задачи решают чат-боты и виртуальные ассистенты в современных веб-приложениях?
5. Какие технологии используются для реализации интеллектуальных чат-ботов и виртуальных ассистентов?
6. В чём заключается роль больших языковых моделей (LLM) в развитии чат-ботов нового поколения?
7. Какие преимущества внедрение чат-ботов даёт бизнесу и пользователям веб-сервисов?
8. Что такое рекомендательные системы и в каких типах веб-приложений они применяются?
9. Какие данные используются для формирования персонализированных рекомендаций пользователям?
10. В чём заключается принцип работы коллаборативной фильтрации в рекомендательных системах?
11. Как контентный подход используется для построения рекомендаций в веб-сервисах?
12. Какие задачи решает искусственный интеллект при анализе поведения пользователей в веб-приложениях?

ГЛАВА 5. Практическая часть

Упражнение 1. Макетирование и прототипирование интерфейса. Личный кабинет агронома

Цель упражнения: Познакомиться с интерфейсом Figma, изучить базовые принципы UI/UX на примере аналогов и создать макет личного кабинета для одного типа пользователя.

Задание:

1. Проанализировать 2-3 аналога (например, личный кабинет банка, панель управления хостингом, дашборд веб-аналитики).
2. Создать в Figma макет личного кабинета агронома.
3. Спроектировать два ключевых экрана: главный экран (дашборд) и экран детальной информации о поле.
4. Подготовить базовую спецификацию для верстальщика.
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

1. Анализ аналогов и определение структуры

1.1 Найдите несколько ресурсов-аналогов, где есть личный кабинет с данными. Примерами таких ресурсов могут быть:

- Веб-аналитика (Яндекс.Метрика). Как они показывают главные цифры? Как устроена навигация?
- Личный кабинет банка (Тинькофф, Сбер): Как отображаются основные счета и операции? Как выглядит карточка детальной информации?
- Панель управления хостингом (cPanel): Как сгруппированы инструменты и информация?

1.2 Внимательно изучите:

- Навигацию. Где находится меню? Как пользователь переходит между разделами?
- Ключевые показатели (KPI): Какие самые важные цифры выносятся на главный экран?

- Карточки: Как оформлены блоки с информацией?

1.3 Сделайте выводы:

Например, вы можете заметить, что у всех аналогов есть боковое меню слева, а самые важные данные (баланс, трафик) вынесены в крупные карточки на главном экране. Это и будет структурой вашего будущего макета.

2. Создание главного экрана (дашборда) в Figma

2.1 Создайте файл и фрейм:

- Откройте Figma, создайте новый файл "Личный кабинет агронома".
- Создайте фрейм для десктопной версии (например, 1440x900 px).

2.2 Спроектируйте боковое меню (sidebar):

- С помощью инструмента "Rectangle" нарисуйте узкую полосу слева (шириной ~250px).
- Добавьте логотип вверху (можно использовать текст "АгроДашборд").
- Ниже перечислите пункты меню: "Главная", "Мои поля", "Отчеты", "Настройки".

2.3 Спроектируйте блок ключевых показателей (KPI):

- Справа от меню, в верхней части экрана, создайте три карточки.
- Карточка 1: "Общая площадь". Внутри крупно напишите "15 500 га".
- Карточка 2: "Текущая урожайность". Внутри "42 ц/га".
- Карточка 3: "Прогноз". Внутри "45 ц/га".

Используйте разные, но неяркие цвета для фона карточек, чтобы они выделялись.

2.4 Спроектируйте список полей:

- Ниже блока KPI создайте таблицу или список из 3-4 элементов.
- Каждый элемент — это "карточка поля". Она должна содержать:
- Название поля (например, "Поле №12").
- Краткую информацию (например, "Краснодарский край | Пшеница").
- Прогноз урожайности ("45 ц/га").

3. Проектирование экрана детальной информации о поле (20-30 минут)

3.1Создайте новый фрейм (можно скопировать размер с предыдущего).

3.2 Создайте заголовок: вверху напишите "Поле №12 | Детальная информация".

3.3 Добавьте вкладки (Tabs): Под заголовком нарисуйте меню вкладок: "Обзор", "История", "Датчики".

3.4 Заполните вкладку "Обзор":

- Текущая культура: Пшеница.
- Прогноз урожайности: Нарисуйте простой линейный график (инструмент "Line").
- Статус полива: Автоматический.

3.5 Заполните вкладку "Датчики":

- Создайте простую таблицу с двумя колонками: "Тип датчика" и "Значение".
- Пример строк: "Влажность почвы | 25%", "Температура | +22°C".

4. Подготовка спецификации и экспорт

4.1 Подготовка спецификации:

- Выберите любой элемент, например, одну из карточек KPI.
- В правой панели Figma перейдите на вкладку Design.
- Скопируйте значения нужных свойств:
  - Width / height: ширина и высота;
  - Fill: цвет заливки;
  - Corner radius: радиус скругления углов;
  - Text properties: шрифт, размер текста, цвет.

4.2 Экспорт макетов:

- Выделите оба фрейма с экранами.
- Нажмите Ctrl+Shift+E (Cmd+Shift+E на Mac).
- Выберите формат PDF и нажмите Export all.

В результате выполнения этого упражнения у вас будет:

1. Простой и понятный макет из двух экранов для личного кабинета агронома.
2. Файл Figma с базовой структурой интерфейса, готовый для передачи в разработку.
3. PDF-файл с макетами для сдачи работы или демонстрации.
4. Практические навыки использования базовых инструментов Figma и понимание того, как анализировать интерфейсы ресурсов аналогов.

Контрольные вопросы к упражнению:

1. В чем разница между фреймом (Frame) и группой (Group/Frame selection) в Figma? Почему для создания каждого экрана необходимо использовать именно фрейм?
2. Использовали ли вы при размещении элементов на экране сетку (Layout Grid) или направляющие? Как это повлияло на аккуратность и выравнивание вашего макета?
3. Объясните, что такое UI Kit и как его создание на отдельном фрейме помогло вам в работе над макетом? Что произойдет, если вам понадобится изменить основной цвет кнопок во всем проекте?
4. Опишите, как пользователь перейдет с главного экрана на экран детальной информации о конкретном поле. Какой элемент интерфейса для этого используется и как он должен работать (в теории, без реализации)?
5. На примере вашего макета объясните простыми словами разницу между UI и UX. Что в вашей работе относится к UI, а что — к UX?

Упражнение 2 . Статическая верстка HTML

Цель упражнения: Освоить базовую верстку веб-страниц, научиться переносить дизайн-макет в HTML-код с использованием семантических тегов.

Задание:

1. Создать структуру HTML-страницы по макету из Figma (хедер, основной контент, футер);
2. Реализовать семантическую разметку (теги <nav>, <section>, <article>);
3. Внедрить формы для ввода данных о полях и культурах;
4. Проверить валидность кода через W3C Validator.
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

1. Подготовка рабочего пространства

1.1 Создайте структуру папок проекта:

В любом удобном месте на компьютере создайте папку, например, agro-dashboard. Внутри нее создайте две подпапки:

- css — здесь будет лежать файл стилей.
- images — для картинок (логотипов и т.д.)

1.2 Создайте основной HTML-файл.

В корневой папке agro-dashboard создайте файл index.html.

2. Создание базовой структуры HTML-документа

Откройте index.html в любом текстовом редакторе (например, VS Code, Sublime Text или даже Блокноте) и напишите стандартный "скелет" HTML5-документа.

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Личный кабинет агронома</title>
    <!-- Подключим CSS-файл -->
    <link rel="stylesheet" href="css/style.css">
</head>
```

```
<body>
    <!-- Здесь будет наш контент -->

    <script src="script.js"></script> <!-- Подключим JS в конце body -->
</body>
</html>
```

- <!DOCTYPE html> - указывает браузеру, что это HTML5;
- <html lang="ru">- корневой тег с указанием языка (русский);
- <head>- содержит метаданные (кодировка, viewport для адаптивности) и ссылку на CSS;
- <body>- основное "тело" страницы, куда мы будем добавлять контент.

3. Верстка хедера (Header)

Согласно нашему макету, в хедере слева находится логотип, а справа — навигация.

```
<body>
    <!-- Header -->
    <header class="header">
        <div class="header__content">
            <!-- Логотип -->
            <a href="#" class="logo">
                <img src="images/logo.png" alt="Логотип АгроДашборд" width="40">
                АгроДашборд
            </a>

            <!-- Навигация -->
            <nav class="nav">
                <ul class="nav__list">
                    <li class="nav__item"><a href="#"
class="nav__link">Главная</a></li>
                    <li class="nav__item"><a href="#" class="nav__link">Мои
поля</a></li>
                    <li class="nav__item"><a href="#"
class="nav__link">Отчеты</a></li>
                    <li class="nav__item"><a href="#"
class="nav__link">Настройки</a></li>
                </ul>
            </nav>
        </div>
    </header>
```

Что мы сделали:

- Использовали семантический тег <header> для шапки сайта.
- Внутри хедера создали контейнер .header__content для выравнивания элементов.
- Логотип обернули в ссылку <a>.
- Для навигации использовали семантический тег <nav>, внутри которого неупорядоченный список <ul> с пунктами меню <li>. Это стандартная и правильная практика для верстки меню.

4. Верстка основного контента (Main)

Основной контент состоит из блока KPI и списка полей. Обернем все это в тег <main>.

```
<main class="main">
    <!-- Блок KPI -->
    <section class="kpi-block">
        <h2 class="visually-hidden">Ключевые показатели (KPI)</h2>
        <div class="kpi-block__cards">
            <article class="kpi-card">
                <h3>Общая площадь</h3>
                <p class="kpi-card__value">15 500 га</p>
            </article>
            <article class="kpi-card">
                <h3>Текущая урожайность</h3>
                <p class="kpi-card__value">42 ц/га</p>
            </article>
            <article class="kpi-card">
                <h3>Прогноз на сезон</h3>
                <p class="kpi-card__value">45 ц/га</p>
            </article>
        </div>
    </section>

    <!-- Список полей -->
    <section class="fields-list">
        <h2 class="visually-hidden">Список полей</h2>
        <div class="fields-list__items">
            <article class="field-card">
                <h3>Поле №12</h3>
                <p><strong>Регион:</strong> Краснодарский край</p>
                <p><strong>Культура:</strong> Пшеница</p>
                <p><strong>Прогноз:</strong> 45 ц/га</p>
            </article>
            <!-- Еще несколько карточек полей -->
            <article class="field-card">
                <h3>Поле №45</h3>
                <p><strong>Регион:</strong> Ростовская обл.</p>
                <p><strong>Культура:</strong> Подсолнечник</p>
                <p><strong>Прогноз:</strong> 32 ц/га</p>
            </article>
        </div>
    </section>
</main>
```

Что мы сделали:

- Обернули основной контент в семантический тег <main>.
- Блоки KPI и список полей обернули в тег <section>, так как это тематические разделы страницы.
- Каждая карточка (KPI или поле) обернута в тег <article>, так как это самодостаточный фрагмент контента.
- Добавили заголовки <h2> с классом visually-hidden. Это техника для доступности: заголовок нужен для скринридеров и SEO, но визуально он скрыт, так как его роль выполняет стилизованный текст в дизайне.

5. Верстка футера (Footer)

В футере обычно размещают информацию о компании или копирайт.

```
    <!-- Footer -->
    <footer class="footer">
        <div class="footer__content">
            <p>&copy; 2026 АгроДашборд. Все права защищены.</p>
        </div>
    </footer>
```

Что мы сделали:

- Использовали семантический тег <footer> для подвала страницы.

6. Внедрение формы для ввода данных

Добавим простую форму для создания нового поля после списка полей.

```
    <section class="add-field-form">
        <h2 class="visually-hidden">Добавить новое поле</h2>
        <form action="#" method="post" class="form">
            <div class="form__row">
                <label for="field-number">Номер поля:</label>
                <input type="text" id="field-number" name="field-number"
required>
            </div>
            <div class="form__row">
                <label for="region">Регион:</label>
                <input type="text" id="region" name="region" required>
            </div>
            <div class="form__row">
                <label for="crop">Культура:</label>
                <select id="crop" name="crop" required>
                    <option value="" disabled selected>Выберите
культуру</option>
                    <option value="wheat">Пшеница</option>
                    <option value="sunflower">Подсолнечник</option>
                    <option value="corn">Кукуруза</option>
                </select>
            </div>
            <button type="submit" class="btn-primary">Добавить поле</button>
        </form>
    </section>
```

Что мы сделали:

- Обернули форму в тег <form>.
- Для каждого поля ввода есть тег <label>, связанный с ним через атрибут for. Это критически важно для доступности.
- Использовали разные типы инпутов: text для номера и региона, select для выбора культуры.
- Добавили атрибуты required для обязательной валидации на стороне браузера.

7. Проверка валидности кода

7.1 Сохраните ваш index.html.

7.2 Откройте W3C Markup Validation Service.

7.3 Перейдите на вкладку "Validate by Direct Input".

7.4 Скопируйте весь код из вашего index.html и вставьте в большое текстовое поле на сайте валидатора.

7.5 Нажмите кнопку "Check".

Если вы следовали инструкциям, валидатор покажет зеленую надпись "Document checking completed. No errors or warnings to show.". Если есть ошибки, валидатор укажет на строку и объяснит, что не так (например, забыли закрыть тег).

В результате выполнения этого упражнения у вас будет:

1. Статическая HTML-страница, структура которой полностью соответствует макету из Figma.
2. Семантически верный код, использующий теги <header>, <nav>, <main>, <section>, <article> и <footer> для логического разделения контента.
3. Валидный HTML-код, успешно прошедший проверку через W3C Validator без ошибок.
4. Функциональная HTML-форма для ввода данных о полях и культурах, готовая к подключению стилей и логики.
5. Фундамент для дальнейшей работы — готовый к стилизации (CSS) и добавлению интерактивности (JavaScript) макет.

Контрольные вопросы к упражнению:

1. В чем разница между тегами <section> и <article>? Почему для блока с карточками полей был выбран именно <article>?
2. Что такое "семантическая верстка" и какие преимущества она дает для поисковых систем (SEO) и для пользователей со специальными потребностями (например, использующих скринридеры)?
3. Опишите правильную структуру HTML-формы. Какие теги являются обязательными для создания функциональной формы?
4. Что такое W3C Validator и зачем проверять код с его помощью? Какие типы ошибок он помогает найти?
5. В чем разница между блочными (, <section>) и строчными (, <a>) элементами? Как это влияет на их отображение на странице?

Упражнение 3 . Интерактивный интерфейс на JavaScript

Цель упражнения: Добавить динамику на фронтенд, научиться работать с JavaScript и популярными библиотеками для создания интерактивных элементов.

Задание:

1. Написать JS-скрипт для валидации форм ввода данных о полях.
2. Реализовать модальные окна (Bootstrap Modals) для просмотра деталей урожая.
3. Создать интерактивную карту полей с помощью Leaflet.js (отметки, всплывающие подсказки).
4. Построить простой график урожайности с помощью Chart.js на основе статических данных.
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

1. Подключение библиотек и подготовка

1.1 Подключите библиотеки:

Убедитесь, что в вашем index.html подключены все необходимые библиотеки. Если вы следовали прошлым шагам, у вас уже есть Bootstrap. Теперь добавим Leaflet и Chart.js. Подключать их лучше в самом конце <body>.

```
<!-- В конце <body> -->
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.j
s"></script>

<!-- Leaflet.js (карты) -->
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.9.4/dist/leaflet.css"/>
<script src="https://unpkg.com/leaflet@1.9.4/dist/leaflet.js"></script>

<!-- Chart.js (графики) -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

<!-- Ваш скрипт -->
<script src="js/script.js"></script>
```

1.2 Создайте файл js/script.js: В папке js создайте файл script.js, куда мы будем писать весь наш код

2. Валидация формы на JavaScript

Мы добавим базовую проверку, чтобы форма не отправлялась с пустыми полями.

2.1 В HTML: Добавьте id к форме и name к полям ввода, если их еще нет.

```
<form id="add-field-form" action="#" method="post" class="form">
    <div class="form__row">
        <label for="field-number">Номер поля:</label>
        <input type="text" id="field-number" name="field-number" required>
    </div>
    ... <!-- другие поля -->
    <button type="submit" class="btn-primary">Добавить поле</button>
</form>
```

2.2 В js/script.js: Напишем обработчик события отправки формы.

```
// script.js

// 1. Валидация формы
document.getElementById('add-field-form').addEventListener('submit',
function(event) {
    // Получаем значения полей
    const fieldNumber = document.getElementById('field-number').value;
    const region = document.getElementById('region').value;
    const crop = document.getElementById('crop').value;

    // Простая проверка: если поле "Номер поля" пустое, отменяем отправку
    if (fieldNumber.trim() === '') {
        alert('Пожалуйста, введите номер поля!'); // Простое сообщение об ошибке
        event.preventDefault(); // Отменяем стандартное действие (отправку
формы)
    }
    // В реальном проекте здесь можно добавить более сложную логику,
    // например, показать ошибку прямо под полем ввода.
});
```

3. Реализация модальных окон (Bootstrap Modals)

Мы сделаем так, чтобы при нажатии на карточку поля открывалось модальное окно с его деталями.

3.1 В HTML: Добавьте код модального окна Bootstrap и кнопки для его открытия.

```
<!-- Кнопка открытия модального окна -->
<button type="button" class="btn btn-primary" data-bs-toggle="modal" data-bs-
target="#fieldModal">
  Просмотреть детали
</button>

<!-- Сам модальный компонент -->
<div class="modal fade" id="fieldModal" tabindex="-1" aria-
labelledby="fieldModalLabel" aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="fieldModalLabel">Детали поля №12</h5>
        <button type="button" class="btn-close" data-bs-dismiss="modal" aria-
label="Close"></button>
      </div>
      <div class="modal-body">
```

```
        <p><strong>Регион:</strong> <span id="modal-region">Краснодарский
край</span></p>
        <p><strong>Культура:</strong> <span id="modal-crop">Пшеница</span></p>
        <p><strong>Прогноз:</strong> <span id="modal-yield">45 ц/га</span></p>
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-bs-
dismiss="modal">Закрыть</button>
      </div>
    </div>
  </div>
</div>
```

3.2 Верстка основного контента (Main)

В js/script.js: Напишем код, который будет заполнять модальное окно данными при клике на карточку.

```
// script.js (продолжение)

// 2. Работа с модальным окном

// Получаем элементы модального окна, куда будем вставлять данные
const modalRegion = document.getElementById('modal-region');
const modalCrop = document.getElementById('modal-crop');
const modalYield = document.getElementById('modal-yield');

// Находим все карточки полей (добавим им класс для удобства)
const fieldCards = document.querySelectorAll('.field-card');

fieldCards.forEach(card => {
  card.addEventListener('click', function() {
    // Извлекаем данные из карточки, на которую кликнули
    const region = card.querySelector('p:nth-of-type(1)
strong').nextSibling.textContent.trim();
    const crop = card.querySelector('p:nth-of-type(2)
strong').nextSibling.textContent.trim();
    const yieldValue = card.querySelector('p:nth-of-type(3)
strong').nextSibling.textContent.trim();

    // Заполняем ими модальное окно
    modalRegion.textContent = region;
    modalCrop.textContent = crop;
    modalYield.textContent = yieldValue;

    // Показываем модальное окно (Bootstrap сам обработает data-bs-toggle)
  });
});
```

4. Интерактивная карта с Leaflet.js

4.1 Мы добавим на страницу карту и отметим на ней наши поля.

```
<section class="map-section">
  <h2 class="visually-hidden">Интерактивная карта полей</h2>
  <div id="mapid" style="height: 400px;"></div> <!-- Контейнер для карты -->
</section>
```

4.2 В js/script.js: Инициализируем карту и добавим метки.

```
<section class="m// script.js (продолжение)

// 3. Интерактивная карта

// Инициализируем карту, привязываем к контейнеру #mapid и задаем начальные
координаты и зум
const map = L.map('mapid').setView([45.022222, 38.977778], 10); // Примерные
координаты Краснодара

// Добавляем тайлы (картинку карты) от OpenStreetMap
L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution: '© OpenStreetMap contributors'
}).addTo(map);

// Добавляем метки на карту с всплывающими подсказками
L.marker([45.04, 38.98]).addTo(map) // Координаты для Поля №12 (примерные)
  .bindPopup('<b>Поле №12</b><br>Культура: Пшеница')
  .openPopup();

L.marker([47.23, 39.70]).addTo(map) // Координаты для Поля №45 (примерные)
  .bindPopup('<b>Поле №45</b><br>Культура: Подсолнечник');ap-section">
  <h2 class="visually-hidden">Интерактивная карта полей</h2>
  <div id="mapid" style="height: 400px;"></div> <!-- Контейнер для карты -->
</section>
```

5. График урожайности с Chart.js

Мы визуализируем данные из карточек KPI в виде графика.

5.1 В HTML: Добавьте контейнер для графика.

```
<section class="chart-section">
  <h2 class="visually-hidden">График урожайности</h2>
  <canvas id="yieldChart"></canvas>
</section>
```

5.2 В js/script.js: Создадим график.Добавим простую форму для создания нового поля после списка полей.

```
// script.js (продолжение)

// 4. График урожайности

// Получаем контекст элемента <canvas>
const ctx = document.getElementById('yieldChart').getContext('2d');

// Создаем новый экземпляр Chart, передавая ему контекст и конфигурацию
const yieldChart = new Chart(ctx, {
  type: 'line', // Тип графика - линейный

  data: {
    labels: ['Апрель', 'Май', 'Июнь', 'Июль', 'Август'], // Подписи оси X
(месяцы)
    datasets: [{
      label: 'Урожайность пшеницы, ц/га', // Название линии на графике
      data: [10, 25, 35, 45, 40], // Статические данные урожайности по месяцам

      // Настройка внешнего вида линии и точек
      borderColor: 'rgb(75, 192, 192)', // Цвет линии
```

```
      backgroundColor: 'rgba(75, 192, 192, 0.2)', // Цвет под линией
(прозрачность)
      tension: 0.1 // Степень кривизны линии (сглаживание)
    }]
  },
  options: {
    responsive: true, // График будет адаптироваться под размер контейнера
    scales: {
      y: {
        beginAtZero: true // Ось Y начинается с нуля
      }
    }
  }
});
```

В результате выполнения этого упражнения у вас будет:

1. Интерактивная веб-страница, которая реагирует на действия пользователя.
2. Валидирующая форма, которая не даст отправить пустые данные.
3. Функционирующее модальное окно, динамически заполняемое контентом.
4. Интерактивная карта с метками и всплывающими подсказками.
5. Визуализированные данные в виде наглядного графика.
6. Практический опыт работы с тремя ключевыми JavaScript-библиотеками, которые широко используются в современной веб-разработке.

Контрольные вопросы к упражнению:

1. Объясните, что делает строка document.getElementById('add-field-form').addEventListener('submit', function(event) { ... }). Разберите её на части: как мы получаем элемент, какое событие отслеживаем и что делает функция-обработчик?
2. Как работает делегирование событий и почему оно может быть полезно? Приведите пример, где вместо того, чтобы вешать обработчик click на каждую карточку поля (.field-card), вы могли бы повесить один обработчик на их общего родителя?
3. В коде Chart.js мы передаем объект с конфигурацией в new Chart(). Назовите два основных раздела этого объекта (data и options) и объясните, за что отвечает каждый из них?

4. Вы заметили, что при быстром клике на кнопку "Добавить поле" форма отправляется несколько раз. Предложите способ, как с помощью JavaScript заблокировать кнопку после первого клика до завершения обработки формы?
5. Как бы вы изменили код для модальных окон, если бы у вас было 10 карточек полей на странице? Почему текущий подход (querySelectorAll('.field-card')) является масштабируемым решением?

Упражнение 4. Верстка с CSS и Bootstrap

Цель упражнения: Научиться стилизовать интерфейс, создавать адаптивную верстку и использовать готовые компоненты фреймворка.

Задание:

1. Подключить CSS и Bootstrap к проекту.
2. Стилизовать макет: цвета, шрифты, отступы согласно UI Kit.
3. Реализовать адаптивную сетку (Bootstrap Grid) для корректного отображения на мобильных устройствах.
4. Оформить карточки культур с использованием компонентов Bootstrap (Cards).
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

Мы будем работать с HTML-структурой, созданной в предыдущем упражнении. Наша задача — "оживить" её с помощью стилей.

1. Подключение Bootstrap и собственного CSS

1.1 Подключите Bootstrap.

В <head> вашего index.html файла, выше подключения вашего собственного CSS файла, добавьте следующую строку. Она подключает минимизированную версию Bootstrap 5.3 (CSS и JS) через CDN (сеть доставки контента).

```
<!-- В <head> -->
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" >

<!-- В конце <body> -->
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.j
s"></script>
```

1.2 Создайте и подключите свой CSS файл.

В папке css создайте файл style.css. В <head> HTML-файла он уже должен быть подключен.

```
<link rel="stylesheet" href="css/style.css">
```

2. Стилизация макета и создание адаптивной сетки

Теперь перейдем к style.css и начнем стилизовать нашу страницу. Главная задача — сделать макет похожим на тот, что был в Figma, и заставить его адаптироваться под разные экраны.

2.1 Сброс стандартных отступов (опционально, но полезно):

В самом начале style.css добавьте:

```
<link rel="stylesheet" href="css/style.css">/* style.css */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
```

2.2 Стилизация хедера и создание сетки:

В Figma у нас был сайдбар (меню) и основной контент. В Bootstrap для этого используется система "Контейнер -> Ряд -> Колонки".

- В HTML: Обернем содержимое <header> в контейнер Bootstrap .container.
- В HTML: Внутри контейнера создадим ряд .row, а в нем две колонки: .col-3 для меню и .col-9 для логотипа (или наоборот, в зависимости от дизайна).

Пример HTML для хедера:

В HTML: Обернем содержимое <header> в контейнер Bootstrap .container.

В HTML: Внутри контейнера создадим ряд .row, а в нем две колонки: .col-3 для меню и .col-9 для логотипа (или наоборот, в зависимости от дизайна).

Пример HTML для хедера:

```
<header class="header">
    <div class="container">
        <div class="row align-items-center">
            <!-- Колонка для логотипа -->
            <div class="col-12 col-md-3">
                <a href="#" class="logo d-flex align-items-center">
                    <img src="images/logo.png" alt="Логотип АгроДашборд"
width="40" class="me-2">
                    <span>АгроДашборд</span>
                </a>
            </div>
            <!-- Колонка для навигации (скрыта на мобильных) -->
            <div class="col-12 col-md-9 d-none d-md-block">
                <nav class="nav justify-content-end">
                    <ul class="nav__list d-flex list-unstyled mb-0">
                        <li class="nav__item"><a href="#"
class="nav__link">Главная</a></li>
                        <li class="nav__item"><a href="#" class="nav__link">Мои
поля</a></li>
                        <li class="nav__item"><a href="#"
class="nav__link">Отчеты</a></li>
                        <li class="nav__item"><a href="#"
class="nav__link">Настройки</a></li>
                    </ul>
                </nav>
```

```
            </div>
        </div>
    </div>
</header>
```

Что мы сделали:

.container - создал адаптивный контейнер с фиксированной шириной на больших экранах.

.row - создал ряд для выравнивания колонок.

.col-12 .col-md-3 - на мобильных (col-12) логотип займет всю ширину. На экранах от md (768px) и выше (col-md-3) он займет 3 колонки из 12.

.d-none .d-md-block - класс Bootstrap для управления видимостью. Навигация будет видна только на средних и больших экранах.

3. Оформление карточек с помощью Bootstrap Card

Вместо того чтобы писать сложные стили для карточек вручную, мы воспользуемся готовым компонентом Bootstrap — Card. Это значительно ускорит процесс.

1. Преобразуем карточки KPI:

Заменим вашу HTML-разметку для KPI на компонент Card.

```
<div class="card text-bg-primary mb-3" style="width: 18rem;">
    <div class="card-body d-flex justify-content-between align-items-center">
        <h5 class="card-title mb-0">Общая площадь</h5>
        <p class="card-text fs-4 mb-0">15 500 га</p>
    </div>
</div>
```

Аналогично поступим с карточками полей.

```
<div class="card mb-3">
    <div class="card-body">
        <h5 class="card-title">Поле №12</h5>
        <p class="card-text"><strong>Регион:</strong> Краснодарский край</p>
        <p class="card-text"><strong>Культура:</strong> Пшеница</p>
        <p class="card-text"><strong>Прогноз:</strong> 45 ц/га</p>
    </div>
</div>
```

4. Финальные штрихи в CSS

Теперь добавим немного кастомных стилей в style.css, чтобы привести цвета и шрифты в соответствие с вашим UI Kit из Figma.

```
/* style.css */

/* --- Цвета из UI Kit --- */
:root {
    --primary-color: #007bff; /* Основной синий */
    --success-color: #28a745; /* Зеленый */
    --warning-color: #ffc107; /* Желтый */
```

```
}
.text-primary { color: var(--primary-color); }
.bg-primary { background-color: var(--primary-color); }
.text-success { color: var(--success-color); }
.bg-success { background-color: var(--success-color); }
.text-warning { color: var(--warning-color); }
.bg-warning { background-color: var(--warning-color); }

/* --- Шрифты и отступы --- */
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
.header {
    background-color: #f8f9fa;
}
.logo img {
    filter: invert(49%) sepia(4%) saturate(492%) hue-rotate(27deg)
brightness(96%) contrast(92%);
}
.nav__link {
    color: #495057;
    margin-left: 20px;
}
.nav__link:hover {
    color: var(--primary-color);
}
```

После выполнения этих шагов у вас будет:

Полностью стилизованная веб-страница, которая визуально соответствует вашему макету из Figma.

Адаптивная верстка, которая корректно отображается как на десктопах, так и на мобильных устройствах благодаря сетке Bootstrap.

Чистый и поддерживаемый код, использующий стандартные компоненты фреймворка, что является индустриальным стандартом.

3. Верстка хедера (Header)

Согласно нашему макету, в хедере слева находится логотип, а справа — навигация.

```
<body>
    <!-- Header -->
    <header class="header">
        <div class="header__content">
            <!-- Логотип -->
            <a href="#" class="logo">
                <img src="images/logo.png" alt="Логотип АгроДашборд" width="40">
                АгроДашборд
            </a>

            <!-- Навигация -->
            <nav class="nav">
                <ul class="nav__list">
                    <li class="nav__item"><a href="#"
class="nav__link">Главная</a></li>
                    <li class="nav__item"><a href="#" class="nav__link">Мои
поля</a></li>
                    <li class="nav__item"><a href="#"
class="nav__link">Отчеты</a></li>
```

```
                    <li class="nav__item"><a href="#"
class="nav__link">Настройки</a></li>
                </ul>
            </nav>
        </div>
    </header>
```

Что мы сделали:

- Использовали семантический тег <header> для шапки сайта.
- Внутри хедера создали контейнер .header__content для выравнивания элементов.
- Логотип обернули в ссылку <a>.
- Для навигации использовали семантический тег <nav>, внутри которого неупорядоченный список <ul> с пунктами меню <li>. Это стандартная и правильная практика для верстки меню.

4. Верстка основного контента (Main)

Основной контент состоит из блока KPI и списка полей. Обернем все это в тег <main>.

```
    <main class="main">
        <!-- Блок KPI -->
        <section class="kpi-block">
            <h2 class="visually-hidden">Ключевые показатели (KPI)</h2>
            <div class="kpi-block__cards">
                <article class="kpi-card">
                    <h3>Общая площадь</h3>
                    <p class="kpi-card__value">15 500 га</p>
                </article>
                <article class="kpi-card">
                    <h3>Текущая урожайность</h3>
                    <p class="kpi-card__value">42 ц/га</p>
                </article>
                <article class="kpi-card">
                    <h3>Прогноз на сезон</h3>
                    <p class="kpi-card__value">45 ц/га</p>
                </article>
            </div>
        </section>

        <!-- Список полей -->
        <section class="fields-list">
            <h2 class="visually-hidden">Список полей</h2>
            <div class="fields-list__items">
                <article class="field-card">
                    <h3>Поле №12</h3>
                    <p><strong>Регион:</strong> Краснодарский край</p>
                    <p><strong>Культура:</strong> Пшеница</p>
                    <p><strong>Прогноз:</strong> 45 ц/га</p>
                </article>
                <!-- Еще несколько карточек полей -->
                <article class="field-card">
                    <h3>Поле №45</h3>
                    <p><strong>Регион:</strong> Ростовская обл.</p>
                    <p><strong>Культура:</strong> Подсолнечник</p>
                    <p><strong>Прогноз:</strong> 32 ц/га</p>
                </article>
            </div>
```

```
        </section>
    </main>
```

Что мы сделали:

- Обернули основной контент в семантический тег <main>.
- Блоки KPI и список полей обернули в тег <section>, так как это тематические разделы страницы.
- Каждая карточка (KPI или поле) обернута в тег <article>, так как это самодостаточный фрагмент контента.
- Добавили заголовки <h2> с классом visually-hidden. Это техника для доступности: заголовок нужен для скринридеров и SEO, но визуально он скрыт, так как его роль выполняет стилизованный текст в дизайне.

5. Верстка футера (Footer)

В футере обычно размещают информацию о компании или копирайт.

```
    <!-- Footer -->
    <footer class="footer">
        <div class="footer__content">
            <p>&copy; 2026 АгроДашборд. Все права защищены.</p>
        </div>
    </footer>
```

Что мы сделали:

- Использовали семантический тег <footer> для подвала страницы.

6. Внедрение формы для ввода данных

Добавим простую форму для создания нового поля после списка полей.

```
    <section class="add-field-form">
        <h2 class="visually-hidden">Добавить новое поле</h2>
        <form action="#" method="post" class="form">
            <div class="form__row">
                <label for="field-number">Номер поля:</label>
                <input type="text" id="field-number" name="field-number"
required>
            </div>
            <div class="form__row">
                <label for="region">Регион:</label>
                <input type="text" id="region" name="region" required>
            </div>
            <div class="form__row">
                <label for="crop">Культура:</label>
                <select id="crop" name="crop" required>
                    <option value="" disabled selected>Выберите
культуру</option>
                    <option value="wheat">Пшеница</option>
                    <option value="sunflower">Подсолнечник</option>
                    <option value="corn">Кукуруза</option>
                </select>
            </div>
            <button type="submit" class="btn-primary">Добавить поле</button>
        </form>
```

```
</section>
```

Что мы сделали:

- Обернули форму в тег <form>.
- Для каждого поля ввода есть тег <label>, связанный с ним через атрибут for. Это критически важно для доступности.
- Использовали разные типы инпутов: text для номера и региона, select для выбора культуры.
- Добавили атрибуты required для обязательной валидации на стороне браузера.

7. Проверка валидности кода

7.1 Сохраните ваш index.html.

7.2 Откройте W3C Markup Validation Service.

7.3 Перейдите на вкладку "Validate by Direct Input".

7.4 Скопируйте весь код из вашего index.html и вставьте в большое текстовое поле на сайте валидатора.

7.5 Нажмите кнопку "Check".

Если вы следовали инструкциям, валидатор покажет зеленую надпись "Document checking completed. No errors or warnings to show.". Если есть ошибки, валидатор укажет на строку и объяснит, что не так (например, забыли закрыть тег).

В результате выполнения этого упражнения у вас будет:

1. Статическая HTML-страница, структура которой полностью соответствует макету из Figma.
2. Семантически верный код, использующий теги <header>, <nav>, <main>, <section>, <article> и <footer> для логического разделения контента.
3. Валидный HTML-код, успешно прошедший проверку через W3C Validator без ошибок.
4. Функциональная HTML-форма для ввода данных о полях и культурах, готовая к подключению стилей и логики.
5. Фундамент для дальнейшей работы — готовый к стилизации (CSS) и добавлению интерактивности (JavaScript) макет.

Контрольные вопросы к упражнению:

1. Объясните, что такое CDN и почему для учебных целей подключение Bootstrap через CDN является удобным решением?

2. Опишите, как работает 12-колоночная сетка Bootstrap. Что означают классы .col-12 и .col-md-3 в контексте адаптивности?
3. Представьте, что вам нужно создать макет из трех равных колонок на десктопе, которые на мобильных устройствах должны выстраиваться в одну колонку. Какие классы Bootstrap вы бы использовали для родительского контейнера и для самих колонок?
4. Перечислите основные структурные элементы компонента Bootstrap Card (.card, .card-body, .card-title и т.д.) и кратко опишите назначение каждого из них.
5. Объясните назначение CSS-свойства box-sizing: border-box;, которое было применено ко всем элементам (*). Как оно влияет на расчет ширины и высоты элементов, особенно при использовании паддингов и бордюров?

Упражнение 5. Проектирование архитектуры веб-приложения

Цель упражнения: Научиться проектировать архитектуру веб-приложения, состоящего из независимых сервисов, включая компоненты искусственного интеллекта.

Задание:

1. Нарисовать UML-диаграмму компонентов системы.
2. Описать API-контракты взаимодействия (REST).
3. Спроектировать микросервисную архитектуру.
4. Обосновать выбор технологий (PostgreSQL, RabbitMQ).
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

Это упражнение можно выполнить в Draw.io или любой другой системе для рисования диаграмм.

1. Проектирование микросервисной архитектуры

Вместо одного большого приложения (монолита), мы разобьем нашу систему на независимые части — микросервисы. Каждый из них решает свою задачу и может обновляться отдельно.

1.1 Выделим сервисы:

- Сервис "Агроном" (Frontend-as-a-Service). Веб-приложение (фронтенд), с которым взаимодействует пользователь.
- API Gateway. Единая точка входа для всех запросов. Проверяет права доступа и направляет запрос в нужный сервис.
- Сервис "Поля" (Fields Service). Управляет данными о полях, культурах, датчиках. Отвечает на запросы от фронтенда.
- Сервис "Аналитика" (Analytics Service). "Тяжелый" сервис. Принимает данные, запускает модели машинного обучения (ML) и возвращает прогноз урожайности.
- Сервис "Чат-бот" (Chatbot Service). Обрабатывает запросы агрономов, использует LLM (большую языковую модель) для генерации ответов.

1.2 Определим взаимодействие (Data Flow).

- Фронтенд запрашивает список полей у Сервиса "Поля".
- Агроном хочет получить прогноз. Фронтенд отправляет запрос в API Gateway, который перенаправляет его в Сервис "Аналитика".
- Сервис "Аналитика", в свою очередь, запрашивает исторические данные у Сервиса "Поля".
- Данные с датчиков в поле не идут напрямую в сервисы. Они попадают в Брокер сообщений (RabbitMQ), откуда их по подписке забирают Сервис "Поля" (для сохранения) и Сервис "Аналитика" (для анализа).

2. Проектирование микросервисной архитектуры

Выбор и обоснование выбора стека технологий.

СУБД PostgreSQL.

Обоснование: Наше приложение работает со структурированными данными (поля, координаты, данные датчиков, пользователи). PostgreSQL — надежная реляционная СУБД. Она отлично поддерживает геоданные (расстояние между точками, нахождение поля в области), что критически важно для работы с пространственными данными. Кроме того, она легко масштабируется и имеет отличную документацию.

Брокер сообщений RabbitMQ.

Обоснование: Данные с тысяч датчиков в поле поступают постоянно и неравномерно (всплески). Если бы сервисы принимали их напрямую, они могли бы не справиться с нагрузкой и "упасть". RabbitMQ выступает в роли буфера: он принимает все сообщения от датчиков и хранит их в очереди. Сервисы "Поля" и "Аналитика" забирают сообщения из очереди тогда, когда готовы их обработать. Это делает систему отказоустойчивой и масштабируемой.

3. Описание API-контрактов (REST API)

Приведите описание того, как сервисы общаются друг с другом через HTTP-запросы. Используйте стандарт REST.

3.1API Сервиса "Поля":

GET /api/fields -> Возвращает список всех полей агронома.

GET /api/fields/{field_id} -> Возвращает детальную информацию о конкретном поле.

GET /api/fields/{field_id}/history -> Возвращает исторические данные по урожайности для этого поля.

3.2 API Сервиса "Аналитика":

POST /api/predict -> Принимает на вход field_id и текущие данные с датчиков. Возвращает прогноз урожайности.

Пример тела запроса:

```
{
  "field_id": 12,
  "soil_moisture": 25,
  "temperature": 22,
  "precipitation_last_week": 15
}
```

Пример ответа:

```
{
  "field_id": 12,
  "predicted_yield": 45.5,
  "confidence": 0.92,
  "prediction_date": "2026-06-17T10:00:00Z"
}
```

3.3 API Сервиса "Аналитика":

POST /api/chat -> Принимает сообщение от пользователя и возвращает ответ бота.

Пример тела запроса:

```
{
  "user_id": 45,
  "message": "Какие удобрения нужны для пшеницы при низкой влажности почвы?"
}
```

Пример ответа:

```
{
  "reply": "При низкой влажности почвы для пшеницы рекомендуется... [совет от
ИИ-модели]"
}
```

4. Нарисовать UML-диаграмму компонентов

Это финальный шаг, который объединяет все вышесказанное в одну картину проектирования.

4.1Компоненты (прямоугольники).

Нарисуйте прямоугольники для каждого сервиса. Агроном (Frontend-приложение), API Gateway (Шлюз API), Сервис "Поля" (Fields Service), Сервис "Аналитика" (Analytics Service), Сервис "Чат-бот" (Chatbot Service), База данных (PostgreSQL). Брокер сообщений (RabbitMQ)

4.2 Соединения (линии со стрелками).

- Нарисуйте сплошную стрелку от "Агронома" к "API Gateway".
- От "API Gateway" нарисуйте сплошные стрелки ко всем остальным сервисам ("Поля", "Аналитика", "Чат-бот").

- Нарисуйте сплошную стрелку от "Аналитики" к "Полям" (для запроса истории).
- Нарисуйте пунктирную стрелку от иконок "Датчики в поле" к RabbitMQ.
- Из RabbitMQ нарисуйте пунктирные стрелки к "Полям" и "Аналитике".
- Нарисуйте сплошные стрелки от сервисов "Поля", "Аналитика" к компоненту "База данных" (PostgreSQL).

4.3 Добавьте легенду.

- пунктирная линия — асинхронное общение через брокер,
- сплошная — синхронный API-запрос

В результате выполнения этого упражнения у вас будет:

1. Визуальная схема (диаграмма), наглядно показывающая все части системы и их взаимодействие.
2. Документ с описанием API, который может служить техническим заданием для команды бэкенда.
3. Обоснованное архитектурное решение, объясняющее выбор ключевых технологий (СУБД, брокер).
4. Понимание того, как разделить сложную задачу на независимые, управляемые сервисы (микросервисный подход).

Контрольные вопросы к упражнению:

1. В чем ключевые отличия монолитной архитектуры от предложенной вами микросервисной?
2. Почему функционал чат-бота вынесен в отдельный сервис (Chatbot Service), а не интегрирован напрямую в основной API? Какие риски мы снижаем таким разделением?
3. Объясните роль RabbitMQ в нашей системе. Что произойдет с данными датчиков, если сервис Fields Service временно недоступен или перегружен?
4. Сравните взаимодействие через REST API (синхронное) и через брокер сообщений (асинхронное). Приведите пример из нашей архитектуры, где

используется каждый из этих подходов, и объясните, почему именно он там уместен.

5. В каких случаях вместо тяжеловесного RabbitMQ можно было бы использовать более простой Redis (в режиме Pub/Sub), а когда его функционала недостаточно? Приведите сценарий из нашей системы?

Упражнение 6. Разработка базы данных и SQL-запросы

Цель упражнения: Научиться проектировать структуру реляционной БД, писать сложные аналитические SQL-запросы, создавать представления (Views) и хранимые процедуры (Stored Procedures).

Задание:

1. Спроектировать физическую модель БД (таблицы fields, crops, sensors, reports).
2. Написать SQL-запросы:
   - Общая площадь посевов по регионам;
   - Топ-5 культур по урожайности за сезон;
   - Средний показатель влажности почвы по датчикам.
3. Создать SQL-представление v_crop_yield.
4. Написать хранимую процедуру get_field_history(field_id).
5. Протестировать на тестовых данных.
6. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

1. Подготовка окружения

Для выполнения этого упражнения вам понадобится доступ к СУБД PostgreSQL. Вы можете использовать:

- Локальную установку (psql в терминале).
- Расширение для VS Code "PostgreSQL".

Подключитесь к вашей базе данных. Для примера создадим базу:

```
CREATE DATABASE agro_db;
\c agro_db; -- Подключаемся к созданной базе
```

2. Проектирование физической модели БД

Спроектируйте четыре основные таблицы. Важно правильно определить связи между ними (один ко многим).

- Таблица fields (Поля): Хранит информацию о каждом физическом поле.

- Таблица crops (Культуры): Справочник всех возможных культур (пшеница, кукуруза и т.д.).
- Таблица sensors (Датчики): Информация об установленных датчиках.
- Таблица reports (Отчеты/Замеры): Основная таблица фактов. Здесь хранятся все замеры с датчиков и данные об урожайности.

Создадим таблицы со связями через внешние ключи (FOREIGN KEY):

```
-- 1. Таблица полей
CREATE TABLE fields (
    field_id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    region VARCHAR(100),
    area_hectares NUMERIC(10, 2), -- Площадь в гектарах
    center_lat DOUBLE PRECISION, -- Координаты центра поля (для карты)
    center_lon DOUBLE PRECISION
);

-- 2. Таблица культур (справочник)
CREATE TABLE crops (
    crop_id SERIAL PRIMARY KEY,
    name VARCHAR(100) UNIQUE NOT NULL,
    scientific_name VARCHAR(255)
);

-- 3. Таблица датчиков
CREATE TABLE sensors (
    sensor_id SERIAL PRIMARY KEY,
    field_id INT REFERENCES fields(field_id) ON DELETE CASCADE,
    type VARCHAR(50), -- 'soil_moisture', 'temperature' и т.д.
    location_description TEXT
);

-- 4. Таблица отчетов (факты)
CREATE TABLE reports (
    report_id BIGSERIAL PRIMARY KEY,
    field_id INT REFERENCES fields(field_id) ON DELETE CASCADE,
    sensor_id INT REFERENCES sensors(sensor_id) ON DELETE SET NULL,
    crop_id INT REFERENCES crops(crop_id) ON DELETE SET NULL,
    measurement_date DATE DEFAULT CURRENT_DATE,
    soil_moisture NUMERIC(5, 2), -- Влажность в %
    temperature NUMERIC(5, 2), -- Температура в °C
    yield_tons_per_ha NUMERIC(7, 2) -- Урожайность в тоннах с гектара
);
```

3. Наполнение тестовыми данными

Прежде чем писать запросы, добавим данные для проверки.

```
-- Добавляем культуры
INSERT INTO crops (name) VALUES ('Пшеница'), ('Подсолнечник'), ('Кукуруза');

-- Добавляем поля
INSERT INTO fields (name, region, area_hectares, center_lat, center_lon)
VALUES
('Поле №12', 'Краснодарский край', 150.5, 45.04, 38.98),
('Поле №45', 'Ростовская область', 200.0, 47.23, 39.70);

-- Добавляем датчики
INSERT INTO sensors (field_id, type, location_description)
VALUES
```

```
(1, 'soil_moisture', 'Северо-западный угол'),
(1, 'temperature', 'Центр поля'),
(2, 'soil_moisture', 'Юго-восток');

-- Добавляем отчеты (замеры)
INSERT INTO reports (field_id, sensor_id, crop_id, measurement_date,
soil_moisture, temperature, yield_tons_per_ha)
VALUES
-- Отчеты для Поля №12 (Пшеница)
(1, 1, 1, '2026-06-10', 25.5, 22.0, NULL),
(1, 2, 1, '2026-06-10', NULL, 22.5, NULL),
(1, 1, 1, '2026-06-17', 24.0, 24.0, NULL),
(1, NULL, 1, '2025-08-20', NULL, NULL, 4.5), -- Итоговая урожайность за прошлый
год

-- Отчеты для Поля №45 (Подсолнечник)
(2, 3, 2, '2026-06-10', 18.0, 26.0, NULL),
(2, NULL, 2, '2025-08-20', NULL, NULL, 2.8); -- Итоговая урожайность за прошлый
год
```

4. Написание SQL-запросов

Теперь решим задачи из задания.

Задача 1. Общая площадь посевов по регионам

```
SELECT
    f.region,
    SUM(f.area_hectares) AS total_area_hectares
FROM fields f
GROUP BY f.region
ORDER BY total_area_hectares DESC;
```

Задача 2. Топ-5 культур по урожайности за последний сезон

Здесь мы агрегируем итоговые данные из отчетов за прошлый год.

```
SELECT
    c.name AS culture,
    AVG(r.yield_tons_per_ha) AS avg_yield
FROM reports r
JOIN crops c ON r.crop_id = c.crop_id
WHERE r.measurement_date BETWEEN '2025-01-01' AND '2025-12-31'
AND r.yield_tons_per_ha IS NOT NULL
GROUP BY c.name
ORDER BY avg_yield DESC
LIMIT 5;
```

Задача 3. Средний показатель влажности почвы по датчикам

```
SELECT
    s.sensor_id,
    s.type,
    AVG(r.soil_moisture) AS avg_soil_moisture
FROM sensors s
LEFT JOIN reports r ON s.sensor_id = r.sensor_id
WHERE s.type = 'soil_moisture'
```

```
GROUP BY s.sensor_id, s.type;
```

5. **Создание SQL-представления (View)**

Представление v_crop_yield будет хранить агрегированные данные по урожайности культур, чтобы фронтенду не приходилось каждый раз выполнять сложный запрос. Фронтенд запрашивает список полей у Сервиса "Поля".

```
-- Создаем представление
CREATE VIEW v_crop_yield AS
SELECT
    c.crop_id,
    c.name AS crop_name,
    COUNT(DISTINCT r.field_id) AS fields_count,
    AVG(r.yield_tons_per_ha) AS average_yield_last_season
FROM crops c
LEFT JOIN reports r ON c.crop_id = r.crop_id
    AND r.measurement_date >= '2025-01-01'
    AND r.measurement_date <= '2025-12-31'
    AND r.yield_tons_per_ha IS NOT NULL
GROUP BY c.crop_id;

-- Теперь можно делать простой запрос к представлению
SELECT * FROM v_crop_yield;
```

6. **Создание SQL-представления (View)**

Процедура get_field_history вернет всю историю урожайности конкретного поля. Используем синтаксис plpgsql.

```
-- Создаем функцию-процедуру
CREATE OR REPLACE FUNCTION get_field_history(p_field_id INT)
RETURNS TABLE(
    report_date DATE,
    yield_tons_per_ha NUMERIC(7,2)
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        r.measurement_date,
        r.yield_tons_per_ha
    FROM reports r
    WHERE r.field_id = p_field_id
      AND r.yield_tons_per_ha IS NOT NULL
    ORDER BY r.measurement_date DESC;
END;
$$ LANGUAGE plpgsql;

-- Вызываем процедуру для поля с ID 1
SELECT * FROM get_field_history(1);
```

В результате выполнения этого упражнения у вас будет:

1. Спроектированная физическая модель БД с четырьмя связанными таблицами.

2. Набор готовых SQL-запросов для решения типовых аналитических задач.
3. Оптимизированное SQL-представление v_crop_yield для быстрого доступа к агрегированным данным.
4. Хранимая процедура get_field_history, инкапсулирующая логику получения истории урожайности.
5. Тестовый набор данных, который подтверждает корректную работу всей написанной логики.

Контрольные вопросы к упражнению:

1. Объясните, что такое нормализация базы данных. К какой нормальной форме (1НФ, 2НФ, 3НФ) вы бы отнесли таблицу reports из нашего примера и почему?
2. Почему для поля area_hectares был выбран тип NUMERIC(10,2), а не INTEGER или FLOAT? В чем преимущества и недостатки этого выбора?
3. Что произойдет, если попытаться выполнить команду DELETE FROM fields WHERE field_id = 1; при наличии записей в таблице reports, связанных с этим полем?
4. Для получения "Топ-5 культур по урожайности" можно было бы использовать как JOIN, так и подзапрос. Опишите, как мог бы выглядеть такой подзапрос, и в чем его потенциальные преимущества или недостатки по сравнению с использованием JOIN?
5. Каковы основные преимущества использования SQL-представления v_crop_yield по сравнению с выполнением исходного сложного запроса каждый раз, когда нужны эти данные? Влияет ли создание VIEW на производительность?

Упражнение 7. Разработка серверной части на FastAPI

Цель упражнения: Научиться создавать REST API для фронтенда и ИИ-модулей.

Задание:

1. Настроить виртуальное окружение и проект FastAPI.
2. Реализовать эндпоинты CRUD для сущностей «Поле», «Культура», «Датчик».
3. Подключить БД через SQLAlchemy ORM.
4. Реализовать эндпоинт /api/predict для передачи данных в модель машинного обучения.
5. Настроить CORS и Swagger-документацию API.
6. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

1. Подготовка окружения и структуры проекта

В корневой папке (agro-dashboard) создайте следующую структуру папок проекта:

```
agro-dashboard/
├── backend/              # Папка для серверной части
│   ├── main.py           # Главный файл приложения
│   ├── models.py         # Модели данных (SQLAlchemy)
│   ├── database.py       # Конфигурация подключения к БД
│   ├── schemas.py        # Схемы для валидации данных (Pydantic)
│   └── crud.py           # Функции для работы с БД (CRUD)
└── requirements.txt      # Файл с зависимостями
```

2. Создайте виртуальное окружение и установите зависимости

В терминале перейдите в папку agro-dashboard и выполните:

```
python -m venv venv          # Создаем виртуальное окружение
source venv/bin/activate     # Активируем его (на Windows:
.\venv\Scripts\activate)

# Устанавливаем необходимые библиотеки
pip install fastapi uvicorn sqlalchemy databases python-multipart

# Для работы с Postgres
pip install asyncpg

# Для валидации данных (Pydantic встроен в FastAPI, но это для ясности)
pip install pydantic
```

```
# Для эндпоинта /predict (заглушка модели)
pip install scikit-learn
```

Содержимое requirements.txt будет:

```
fastapi
uvicorn
sqlalchemy
databases
python-multipart
asyncpg
scikit-learn
```

3. Настройка подключения к БД (database.py)

Создайте файл backend/database.py. Здесь мы определим URL для подключения и объект Database для асинхронной работы.

```
# backend/database.py

import os
from databases import Database

# URL для подключения к БД. Данные лучше брать из переменных окружения.
# Для локальной разработки можно указать прямо.
DATABASE_URL = os.getenv("DATABASE_URL",
"postgresql://user:password@localhost/agro_db")

# Создаем объект Database
database = Database(DATABASE_URL)
```

4. Создание моделей данных с SQLAlchemy (models.py)

Здесь мы опишем структуру наших таблиц, используя SQLAlchemy.

```
# backend/models.py

from sqlalchemy import (
    Column, Integer, String, Float, Date, ForeignKey, Numeric, BigInteger
)
from sqlalchemy.orm import relationship
from .database import Base

# Модель для таблицы 'fields'
class Field(Base):
    __tablename__ = "fields"

    field_id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    region = Column(String, index=True)
    area_hectares = Column(Numeric(10,2))
    center_lat = Column(Float)
    center_lon = Column(Float)

    # Связь с датчиками (один ко многим)
    sensors = relationship("Sensor", back_populates="field")

# Модель для таблицы 'crops'
```

```
class Crop(Base):
    __tablename__ = "crops"

    crop_id = Column(Integer, primary_key=True, index=True)
    name = Column(String, unique=True, index=True)
    scientific_name = Column(String)

# Модель для таблицы 'sensors'
class Sensor(Base):
    __tablename__ = "sensors"

    sensor_id = Column(Integer, primary_key=True, index=True)
    field_id = Column(Integer, ForeignKey("fields.field_id"))
    type = Column(String) # e.g., 'soil_moisture', 'temperature'
    location_description = Column(String)

    # Связь с полем (многие к одному)
    field = relationship("Field", back_populates="sensors")
```

5. Создание схем для валидации (schemas.py)

FastAPI использует Pydantic для валидации входящих запросов и сериализации ответов.

```
# backend/schemas.py

from pydantic import BaseModel, ConfigDict, Field

# Схемы для сущности "Поле"
class FieldBase(BaseModel):
    name: str
    region: str
    area_hectares: float | None = None

class FieldCreate(FieldBase):
    pass

class Field(FieldBase):
    field_id: int

    class Config:
        orm_mode = True # Позволяет Pydantic работать с ORM-объектами

# Схемы для сущности "Культура"
class CropBase(BaseModel):
    name: str

class CropCreate(CropBase):
    pass

class Crop(CropBase):
    crop_id: int

    class Config:
        orm_mode = True

# Схема для эндпоинта предсказания
class PredictRequest(BaseModel):
    model_config = ConfigDict(
        extra="forbid",
        allow_inf_nan=False,
```

```
    )

    field_id: int = Field(gt=0)
    soil_moisture: float = Field(ge=0, le=100)
    temperature: float
    precipitation_last_week: float = Field(ge=0)
```

6. Реализация CRUD-операций (crud.py)

Здесь пишутся функции для создания, чтения, обновления и удаления данных.

```
# backend/crud.py

from sqlalchemy.orm import Session, selectinload
from . import models, schemas

# --- Операции для "Полей" ---
def get_fields(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Field).offset(skip).limit(limit).all()

def create_field(db: Session, field: schemas.FieldCreate):
    db_field = models.Field(**field.dict())
    db.add(db_field)
    db.commit()
    db.refresh(db_field) # Обновляем объект, чтобы получить ID после commit()
    return db_field

# --- Операции для "Культур" ---
def get_crops(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Crop).offset(skip).limit(limit).all()

def create_crop(db: Session, crop: schemas.CropCreate):
    db_crop = models.Crop(**crop.dict())
    db.add(db_crop)
    db.commit()
    db.refresh(db_crop)
    return db_crop
```

Продолжите код для других сущностей (Sensor) ...

7. Создание главного файла приложения (main.py)

Здесь мы должны определить эндпоинты. Поскольку реальной модели у нас пока еще нет, то прописываем в коде заглушки, для того, чтобы затем доработать.

```
# backend/main.py

from fastapi import FastAPI, Depends, HTTPException, status, BackgroundTasks
from fastapi.middleware.cors import CORSMiddleware # Для CORS
import joblib # Для загрузки модели ML (если бы она была)
import numpy as np

from .database import database, engine # Подключение к БД (из Шага 2)
from . import models, schemas, crud # Импортируем наши модули
from sqlalchemy.orm import Session, sessionmaker # Для работы с сессиями БД

# Создаем таблицы в БД (выполняется один раз при старте)
models.Base.metadata.create_all(bind=engine)
```

```
app = FastAPI(title="Агро-Дашборд API")

# --- Настройка CORS ---
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Разрешаем все домены.
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Зависимость для получения сессии БД (упрощенный вариант)
def get_db():
    try:
        db = Session(engine)
        yield db
        db.close()

# --- Эндпоинты CRUD для "Полей" ---
@app.get("/api/fields/", response_model=list[schemas.Field])
async def read_fields(skip: int = 0, limit: int = 100, db: Session =
Depends(get_db)):
    return crud.get_fields(db, skip=skip, limit=limit)

@app.post("/api/fields/", response_model=schemas.Field,
status_code=status.HTTP_201_CREATED)
async def create_field(field: schemas.FieldCreate, db: Session =
Depends(get_db)):
    return crud.create_field(db, field=field)

# --- Эндпоинты CRUD для "Культур" ---
@app.get("/api/crops/", response_model=list[schemas.Crop])
async def read_crops(skip: int = 0, limit: int = 100, db: Session =
Depends(get_db)):
    return crud.get_crops(db, skip=skip, limit=limit)

@app.post("/api/crops/", response_model=schemas.Crop,
status_code=status.HTTP_201_CREATED)
async def create_crop(crop: schemas.CropCreate, db: Session = Depends(get_db)):
    return crud.create_crop(db, crop=crop)

# --- Эндпоинт для предсказания урожайности ---
@app.post("/api/predict/", response_model=schemas.PredictResponse)
async def predict_yield(request: schemas.PredictRequest):
    # Здесь должна быть логика работы с моделью ML.
    # Так как у нас нет пока реальной модели, создадим заглушку.

    # В реальном проекте здесь было бы:
    # model = joblib.load('crop_yield_model.pkl')
    # prediction = model.predict(np.array([[request.soil_moisture or 0,
request.temperature or 0]]))

    # Заглушка: просто возвращаем среднее значение на основе field_id (для
примера)
    dummy_prediction = {12: 45.5, 45: 32.0}.get(request.field_id, None)

    if dummy_prediction is None:
        raise HTTPException(status_code=404, detail="Field not found for
prediction")

    return {"predicted_yield": dummy_prediction}
```

8. Запуск сервера и проверка Swagger-документации

8.1 Запустите сервер Uvicorn из корня проекта (agro-dashboard):

```
uvicorn backend.main:app --reload --host 0.0.0.0 --port 8000
```

8.2 Проверьте Swagger-документацию: Откройте в браузере

```
http://127.0.0.1:8000/docs
```

8.3 Протестируйте эндпоинт: Перейдите на вкладку POST /api/fields/. Вы увидите интерактивную форму для создания нового поля. Попробуйте отправить JSON-запрос

```
{
  "name": "Новое поле",
  "region": "Ставропольский край",
  "area_hectares": 120.5,
  "center_lat": 45.5,
  "center_lon": 42.0
}
```

Нажмите "Execute" — если всё сделано верно, вы получите ответ с созданным объектом поля.

Протестируйте /api/predict/: Перейдите на вкладку POST /api/predict/ и отправьте запрос:

```
{
  "field_id": 12,
  "soil_moisture": 25,
  "temperature": 22,
  "precipitation_last_week": 15
}
```

Вы должны получить ответ с предсказанным значением урожайности.

В результате выполнения этого упражнения у вас будет:

1. Работающий REST API на FastAPI.
2. Полностью задокументированное API с интерактивной Swagger-панелью.
3. Готовые эндпоинты CRUD для управления данными через фронтенд.
4. Эндпоинт /api/predict, который готов к интеграции с реальной моделью машинного обучения.
5. Настроенный CORS, позволяющий вашему фронтенду (даже если он запущен на другом порту) безопасно обращаться к бэкенду.

Контрольные вопросы к упражнению:

1. Что такое Depends() в FastAPI и как оно используется в коде (например, db: Session = Depends(get_db))? Какие преимущества дает такой подход при проектировании API?
2. Зачем в нашем приложении была настроена политика CORS с помощью CORSMiddleware? Что бы произошло, если бы мы забыли ее настроить, и фронтенд с портом 3000 пытался бы обратиться к API на порту 8000?
3. REST и CRUD: Опишите, как принципы CRUD (Create, Read, Update, Delete) реализованы в REST API нашего приложения.
4. Какую роль играют Pydantic-модели (например, FieldCreate) в эндпоинтах FastAPI?
5. В чем разница между возвратом объекта Pydantic-модели (например, return db_field) и использованием JSONResponse или Response напрямую?

Упражнение 8. Модуль предиктивной аналитики урожайности

Цель упражнения: Внедрить модель машинного обучения в веб-приложение.

Задание:

1. Подготовить датасет и обучить модель (в Jupyter Notebook).
2. Сохранить обученную модель в файл (joblib/pickle).
3. Загрузить модель в FastAPI и реализовать логику предсказания.
4. Вывести прогноз на фронтенд.
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

1. Подготовка данных и обучение модели

Этот этап выполняется отдельно от основного проекта, обычно в среде анализа данных, будем использовать для этого Jupyter Notebook.\

1.1. Создание датасета

Для начала нам нужны данные. В реальном проекте вы бы использовали исторические данные из вашей базы данных. Для примера создадим небольшой синтетический датасет прямо в коде.

```
# Создаем синтетические данные для наглядности
import pandas as pd
import numpy as np

np.random.seed(42) # Для воспроизводимости результата
n_samples = 500

# Создаём признаки.
data = {
    "soil_moisture": np.random.uniform(15, 35, n_samples),          # %
    "temperature": np.random.normal(22, 3, n_samples),            # °C
    "precipitation_last_week": np.random.uniform(0, 50, n_samples),  # мм
}
# Вычисляем целевую переменную после создания словаря.
# Формула условная и не является агрономической моделью.
data["yield_tons_per_ha"] = (
    30
    + 0.8 * data["soil_moisture"]
    - 1.5 * np.abs(data["temperature"] - 22)
    + np.random.normal(0, 1, n_samples)
)
```

```
# Формируем таблицу.
df = pd.DataFrame(data)

# Сохраним в CSV, чтобы имитировать загрузку из файла
df.to_csv('crop_yield_dataset.csv', index=False)

python
from pathlib import Path

import joblib
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split

# Признаки и целевая переменная.
feature_columns = [
    "soil_moisture",
    "temperature",
    "precipitation_last_week",
]

X = df[feature_columns]
y = df["yield_tons_per_ha"]

# Разделяем данные до обучения.
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42,
)

# Обучаем модель.
model = RandomForestRegressor(
    n_estimators=200,
    random_state=42,
    n_jobs=1,
)
model.fit(X_train, y_train)

# Оцениваем качество на тестовой выборке.
y_pred = model.predict(X_test)

print(f"MAE: {mean_absolute_error(y_test, y_pred):.3f}")
print(f"R²: {r2_score(y_test, y_pred):.3f}")

# Сохраняем обученную модель рядом с backend/main.py.
model_path = Path("backend") / "crop_yield_model.pkl"
model_path.parent.mkdir(parents=True, exist_ok=True)

joblib.dump(model, model_path)
print(f"Модель сохранена: {model_path.resolve()}")
```

Результат этого шага - файл crop_yield_model.pkl, который содержит обученную модель. Значения MAE и R^2 рассчитываются на тестовой выборке. Поскольку данные синтетические, полученные показатели характеризуют воспроизведение условной зависимости, а не качество прогноза реальной урожайности.

2. Интеграция модели в FastAPI

2.1 Добавьте scikit-learn и joblib в список зависимостей вашего бэкенда, если их там еще нет.

Файл backend/requirements.txt:

```
fastapi
uvicorn
sqlalchemy
databases
python-multipart
asyncpg
scikit-learn
joblib
```

Установите зависимости.

```
pip install -r backend/requirements.txt
```

2.2. Загрузка модели и реализация эндпоинта

Откройте файл backend/main.py. Мы будем работать внутри функции predict_yield.

```
backend/main.py
from fastapi import FastAPI, Depends, HTTPException, status
import joblib
import numpy as np
# ... другие импорты ...
app = FastAPI(title="Агро-Дашборд API")
# --- ГЛОБАЛЬНАЯ ПЕРЕМЕННАЯ ДЛЯ МОДЕЛИ ---
# Модель загружается один раз при старте приложения Uvicorn.
# path/to/model зависит от структуры вашего проекта.
try:
    crop_yield_model = joblib.load("backend/crop_yield_model.pkl")
except Exception as e:
    print(f"Warning: Model not loaded. {e}")
    crop_yield_model = None

# ... остальной код (CRU D-эндпоинты, CORS) ...

@app.post("/api/predict/", response_model=schemas.PredictResponse)
async def predict_yield(request: schemas.PredictRequest):
    """
    Эндпоинт для получения прогноза урожайности.
```

```
    """
    if crop_yield_model is None:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Predictive model is not available."
        )
    # 1. Подготовка входных данных для модели
    # Модель ожидает на входе двумерный массив [[val1, val2, val3]]
    input_data = np.array([
        [
            request.soil_moisture or 0, # Используем 0, если значение не передано
            request.temperature or 0
            # Здесь могли бы быть и другие признаки
        ]
    ])
    # 2. Получение прогноза
    prediction = crop_yield_model.predict(input_data)[0] # [0], так как предсказание одно

    return {"predicted_yield": float(prediction)}
```

Теперь ваш эндпоинт /api/predict/ использует реальную ML-модель вместо заглушки.

3. Отображение прогноза на фронтенде

Предположим, у вас на странице есть форма для ввода данных и блок для вывода графика.

3.1. HTML-разметка (frontend/index.html)

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Агро-Дашборд</title>
    <!-- Подключаем Chart.js -->
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
<body>
    <h1>Прогноз урожайности</h1>

    <form id="prediction-form">
        <label for="field-id">ID Поля:</label>
        <input type="number" id="field-id" name="field_id" required><br><br>

        <label for="soil-moisture">Влажность почвы (%):</label>
        <input type="number" step="0.1" id="soil-moisture"
name="soil_moisture"><br><br>
```

```
        <label for="temperature">Температура (°C):</label>
        <input type="number" step="0.1" id="temperature"
name="temperature"><br><br>

        <button type="submit">Получить прогноз</button>
    </form>

    <div style="margin-top: 20px;">
        <canvas id="yieldChart" width="400" height="200"></canvas>
    </div>

    <script src="static/app.js"></script> <!-- Наш JS-файл -->
</body>
</html>
```

3.2. JavaScript-код (frontend/static/app.js)

Этот скрипт будет отправлять данные на сервер и рисовать график

```
// frontend/static/app.js

const form = document.getElementById('prediction-form');
const ctx = document.getElementById('yieldChart').getContext('2d');
let yieldChart;

form.addEventListener('submit', async function(event) {
    event.preventDefault(); // Предотвращаем перезагрузку страницы

    const formData = new FormData(form);
    const payload = {};
    // Преобразуем FormData в обычный объект
    for(let [key, value] of formData.entries()) {
        payload[key] = parseFloat(value); // Приводим строки к числам
    }

    try {
        // Отправляем POST-запрос на наш API
        const response = await fetch('/api/predict/', {
            method: 'POST',
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify(payload)
        });

        if (!response.ok) {
            throw new Error(`Ошибка сервера: ${response.status}`);
        }

        const result = await response.json();
        const predictedYield = result.predicted_yield;

        console.log('Прогноз:', predictedYield);

        // Рисуем или обновляем график
        renderChart(predictedYield);

    } catch (error) {
        console.error("Не удалось получить прогноз:", error);
        alert("Произошла ошибка при запросе к серверу.");
    }
});

function renderChart(predictedValue) {
    // Данные для сравнения с прошлым годом (статичные для примера)
```

```
    const lastYearYield = 42.5; // т/га

    // Если график уже существует, уничтожаем его перед перерисовкой
    if (yieldChart) {
        yieldChart.destroy();
    }

    yieldChart = new Chart(ctx, {
        type: 'bar',
        data: {
            labels: ['Текущий год (прогноз)', 'Прошлый год'],
            datasets: [{
                label: 'Урожайность (т/га)',
                data: [predictedValue, lastYearYield],
                backgroundColor: [
                    '#4CAF50', // Зеленый для прогноза
                    '#FFC107'  // Желтый для прошлого года
                ]
            }]
        },
        options: {
            responsive: true,
            scales: {
                y: {
                    beginAtZero: true,
                    title: { display: true, text: 'Урожайность (т/га)' }
                }
            },
            plugins: {
                title: {
                    display: true,
                    text: 'Сравнение урожайности'
                }
            }
        }
    });
}
```

После выполнения всех шагов ваше приложение теперь работает следующим образом:

1. Пользователь вводит данные (например, влажность и температуру) во фронтенд-форме.
2. JavaScript отправляет эти данные на эндпоинт /api/predict/ вашего FastAPI-сервиса.
3. Бэкенд загружает сохраненный .pkl файл с моделью RandomForest, передает ему полученные данные и получает числовой прогноз.
4. Прогноз возвращается на фронтенд в виде JSON.
5. JavaScript на фронтенде принимает это число и визуализирует его на графике вместе с данными за прошлый год, предоставляя пользователю понятную аналитику.

Контрольные вопросы к упражнению:

1. Опишите пошагово процесс обучения модели в Jupyter Notebook. Какую роль играют тренировочная (X_train) и тестовая (X_test) выборки, и почему важно их разделять?
2. Где должен находиться файл crop_yield_model.pkl относительно кода бэкенда при локальной разработке? Как изменится способ загрузки файла при развертывании приложения в Docker-контейнере или облачной среде (например, AWS)?
3. Почему для отображения сравнения прогноза с прошлым годом был выбран Chart.js? Какие альтернативы (например, D3.js или встроенные SVG-элементы) существуют и в каких случаях они могли бы быть более предпочтительны?
4. На графике сравнивается прогноз текущего года и статичное значение прошлого года. Предложите модификацию фронтенд-кода, чтобы динамически получать данные за прошлый год из вашего же REST API, а не использовать жесткое значение (const lastYearYield = 42.5;)
5. Почему перед передачей данных в метод .predict() необходимо преобразовать их в двумерный массив NumPy (np.array([[...]])), даже если на вход подается всего одна запись?
6. Что означает запись depends_on и гарантирует ли она, что сервис-источник уже полностью готов к работе на момент запуска зависимого сервиса

Упражнение 9. Интеграция бэкенда и фронтенда

Цель упражнения: Обеспечить взаимодействие клиентской и серверной частей приложения.

Задание:

1. Настроить асинхронные запросы (fetch/axios) из JS к API бэкенда.
2. Реализовать загрузку данных на страницу (например, списка полей).
3. Реализовать отправку данных на сервер (например, создание нового поля).
4. Обработать возможные ошибки сети и некорректные ответы сервера.
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

Мы будем использовать уже созданные нами компоненты:

- Бэкенд. FastAPI с эндпоинтами /api/fields/ (GET для получения списка, POST для создания).
- Фронтенд. Статический HTML-файл с подключенным JavaScript.

1. Подготовка фронтенда

Убедитесь, что у вас есть frontend/index.html и frontend/static/app.js.

В index.html создадим простую структуру для отображения данных:

```
<!-- frontend/index.html -->
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Интеграция с API</title>
    <style>
        .error { color: red; }
        .success { color: green; }
    </style>
</head>
<body>
    <h1>Поля</h1>
    <div id="status-message"></div>
    <ul id="fields-list">
        <!-- Поля будут загружены сюда -->
    </ul>

    <h2>Добавить новое поле</h2>
```

```
    <form id="add-field-form">
        <label>Название: <input type="text" id="field-name"
required></label><br>
        <label>Регион: <input type="text" id="field-region"
required></label><br>
        <button type="submit">Создать</button>
    </form>

    <script src="static/app.js"></script>
</body>
</html>
```

2. Написание асинхронного JS-кода (fetch)

Теперь в frontend/static/app.js напишем логику для взаимодействия с API.

```
// frontend/static/app.js

// --- 1. Константы и элементы DOM ---
const API_URL = 'http://localhost:8000/api'; // Адрес нашего FastAPI-сервиса
const fieldsList = document.getElementById('fields-list');
const addFieldForm = document.getElementById('add-field-form');
const statusMessage = document.getElementById('status-message');

// --- 2. Функция для загрузки списка полей ---
async function loadFields() {
    try {
        const response = await fetch(`${API_URL}/fields/`);

        // --- Обработка ошибок HTTP-статуса ---
        if (!response.ok) {
            // Если статус не 2xx (например, 404, 500)
            throw new Error(`Ошибка сети: ${response.status}
${response.statusText}`);
        }

        const fields = await response.json(); // Парсим JSON-ответ

        // Очищаем список перед загрузкой
        fieldsList.innerHTML = '';

        // Отображаем каждое поле в списке
        fields.forEach(field => {
            const li = document.createElement('li');
            li.textContent = `ID: ${field.field_id}, Название: ${field.name},
Регион: ${field.region}`;
            fieldsList.appendChild(li);
        });

        showMessage('Список полей успешно загружен.', 'success');

    } catch (error) {
        // --- Обработка сетевых ошибок (нет ответа от сервера) и ошибок из
блока if ---
        console.error("Не удалось загрузить поля:", error);
        showMessage('Ошибка при загрузке данных с сервера. Проверьте консоль.',
'error');
    }
}

// --- 3. Функция для отправки формы (создание нового поля) ---
addFieldForm.addEventListener('submit', async function(event) {
    event.preventDefault(); // Предотвращаем перезагрузку страницы
```

```
    const fieldData = {
        name: document.getElementById('field-name').value,
        region: document.getElementById('field-region').value
    };

    try {
        const response = await fetch(`${API_URL}/fields/`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(fieldData)
        });

        if (!response.ok) {
            // Если сервер вернул ошибку валидации или другую проблему
            const errorData = await response.json();
            throw new Error(`Ошибка сервера: ${errorData.detail ||
response.statusText}`);
        }

        const newField = await response.json();
        showMessage(`Поле "${newField.name}" успешно создано!`, 'success');

        // Обновляем список полей, чтобы увидеть новый элемент
        loadFields();

    } catch (error) {
        console.error("Ошибка при создании поля:", error);
        showMessage(error.message || 'Произошла неизвестная ошибка.', 'error');
    }
});

// --- Вспомогательная функция для вывода сообщений ---
function showMessage(text, type) {
    statusMessage.textContent = text;
    statusMessage.className = type; // Применяем класс 'success' или 'error'
}

// --- Запускаем загрузку данных при открытии страницы ---
document.addEventListener('DOMContentLoaded', loadFields);
```

3. Запуск приложения и проверка

Убедитесь, что ваш FastAPI сервер запущен (uvicorn backend.main:app --reload).

Откройте frontend/index.html в браузере.

Что должно произойти:

- При загрузке страницы вы увидите сообщение "Список полей успешно загружен." и список полей из вашей базы данных.
- Заполните форму "Добавить новое поле" и нажмите "Создать".
- Если все верно, появится сообщение об успехе, и новый элемент добавится в список без перезагрузки страницы.

Тестирование обработки ошибок:

- Попробуйте остановить сервер FastAPI и обновить страницу. Вы должны увидеть сообщение об ошибке сети.
- Попробуйте отправить форму с пустыми полями (если убрать атрибут required). Сервер вернет ошибку валидации, которую наш код отловит и выведет на экран.

После выполнения всех шагов ваше приложение теперь работает следующим образом:

1. Фронтенд динамически загружает данные с сервера при открытии.
2. Пользователь может отправлять данные на сервер через форму.
3. Интерфейс обновляется в реальном времени без перезагрузки страницы.
4. Система корректно обрабатывает и показывает пользователю различные типы ошибок (проблемы с сетью, ошибки валидации на сервере).

Контрольные вопросы к упражнению:

1. Объясните разницу между синхронным и асинхронным выполнением кода в контексте веб-браузера?
2. В чем разница между ошибкой, вызванной !response.ok (проблема на стороне сервера, например, 404 или 500), и ошибкой, пойманной блоком catch (например, сеть недоступна)? Какую информацию о проблеме может предоставить каждый из этих случаев?
3. Опишите полный цикл отправки формы "Добавить новое поле". Какие шаги происходят в коде JavaScript от момента нажатия кнопки "Создать" до обновления списка полей на странице?

Упражнение 10. Контейнеризация приложения с помощью Docker

Цель упражнения: Научиться упаковывать приложение в изолированные контейнеры для удобства разработки, тестирования и развертывания.

Задание:

1. Создать Dockerfile для бэкенда (FastAPI).
2. Создать Dockerfile для фронтенда (Nginx).
3. Настроить docker-compose.yml для оркестрации всех сервисов.
4. Запустить все приложение командой docker-compose up.
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

1. Подготовка проекта

Убедитесь, что у вас установлены Docker Engine и Docker Compose. Структура вашего проекта должна выглядеть примерно так:

```
agro-dashboard/
├── backend/
│   ├── main.py
│   ├── requirements.txt
│   └── ...
├── frontend/  <-- Папка с собранным фронтендом (HTML, CSS, JS)
│   ├── index.html
│   ├── static/
│   └── ...
└── docker/
    ├── Dockerfile.backend
    ├── Dockerfile.frontend
    └── docker-compose.yml
```

Папка frontend/ должна содержать уже собранные статические файлы. Если вы верстали с помощью HTML/CSS/JS, просто скопируйте их туда. Если бы вы использовали фреймворк (например, React), вы бы сначала выполнили команду сборки (npm run build), а затем скопировали содержимое папки build/ в папку frontend/

2. Создание Dockerfile для бэкенда (FastAPI)

Создайте файл docker/Dockerfile.backend. Этот файл содержит инструкции для сборки образа нашего API-сервера:

```
# docker/Dockerfile.backend

# 1. Определяем базовый образ (легковесный Python)
FROM python:3.11-slim

# 2. Устанавливаем системные зависимости (нужны для сборки некоторых Python-
пакетов)
RUN apt-get update && apt-get install -y --no-install-recommends gcc

# 3. Создаем рабочую директорию внутри контейнера
WORKDIR /app

# 4. Копируем файл с зависимостями и устанавливаем их
COPY ./backend/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 5. Копируем весь код бэкенда в контейнер
COPY ./backend/ .

# 6. Открываем порт, на котором будет работать Uvicorn
EXPOSE 8000

# 7. Задаем команду для запуска контейнера
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

3. Создание Dockerfile для фронтенда (Nginx)

Создайте файл docker/Dockerfile.frontend. Этот образ будет просто отдавать статические файлы.

```
# docker/Dockerfile.frontend

# 1. Используем официальный легковесный образ Nginx
FROM nginx:stable-alpine

# 2. Удаляем дефолтную конфигурацию Nginx
RUN rm /etc/nginx/conf.d/default.conf

# 3. Копируем нашу конфигурацию Nginx (создадим ее на следующем шаге)
COPY ./docker/nginx.conf /etc/nginx/conf.d/

# 4. Копируем все собранные файлы фронтенда в папку, которую будет обслуживать
 Nginx
COPY ./frontend/ /usr/share/nginx/html

# 5. Открываем стандартный HTTP-порт
EXPOSE 80
```

Теперь создадим файл конфигурации Nginx docker/nginx.conf.

```
# docker/nginx.conf

server {
    listen 80;
    server_name localhost;

    # Указываем корневую директорию для статических файлов
    root /usr/share/nginx/html;
    index index.html;

    location / {
```

```
        # Пробуем отдать файл напрямую, если он существует
        try_files $uri $uri/ /index.html;
    }
```

Эта конфигурация нужна для поддержки "single-page applications" (SPA).

Она говорит Nginx: если запрашиваемый файл не найден, отдай index.html.

4. Создание docker-compose.yml

Это главный файл, который объединит все сервисы в одну сеть и запустит их вместе.

Создайте файл docker/docker-compose.yml:

```
# docker/docker-compose.yml

version: '3.8'
services:
  # --- Сервис Базы Данных (PostgreSQL) ---
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: agro_db          # Имя БД (как в database.py)
      POSTGRES_USER: user           # Пользователь БД
      POSTGRES_PASSWORD: password  # Пароль БД
    volumes:
      - pg_data:/var/lib/postgresql/data # Для сохранения данных БД

  # --- Сервис Бэкенда (FastAPI) ---
  backend:
    build:
      context: .
      dockerfile: ./docker/Dockerfile.backend
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/agro_db # Важно! db -
это имя сервиса!
    depends_on:
      - db # Сначала поднимется сервис db, потом backend
    ports:
      - "8000:8000" # Пробрасываем порт наружу

  # --- Сервис Фронтенда (Nginx) ---
  frontend:
    build:
      context: .
      dockerfile: ./docker/Dockerfile.frontend
    ports:
      - "80:80" # Пробрасываем порт наружу (основной сайт)
    depends_on:
      - backend # Фронтенд ждет, пока поднимется бэкенд

volumes:
  pg_data: # Определение именованного volume для данных БД

networks:
  default:
    external: false # Используем дефолтную сеть, которую создаст Docker Compose
```

Что мы сделали:

- Описали три сервиса: db, backend, frontend.
- Для БД используем готовый образ postgres. С помощью переменных окружения создаем базу, пользователя и пароль. С помощью volumes сохраняем данные БД, чтобы они не пропали при перезапуске контейнера.
- Для бэкенда указываем, какой Dockerfile использовать (build). Передаем в контейнер переменную DATABASE_URL, где в качестве хоста указано имя сервиса БД (db). Это позволяет контейнерам общаться друг с другом по именам сервисов внутри сети Docker. Пробрасываем порт 8000.
- Для фронтенда указываем его Dockerfile. Пробрасываем порт 80. С помощью depends_on гарантируем порядок запуска.
- Сети и Volumes: Определили именованный том для данных БД.

5. Запуск приложения

Откройте терминал в корневой папке проекта (agro-dashboard). Убедитесь, что вы находитесь в глобальной среде Docker (не в виртуальном окружении Python!). Выполните команду:

```
docker compose up --build -d
```

Откройте браузер и перейдите по адресу http://localhost/. Вы должны увидеть главную страницу вашего фронтенда.

Осуществите проверку API. Откройте Swagger-документацию по адресу http://localhost:8000/docs. Вы можете протестировать эндпоинты API прямо из браузера.

Чтобы остановить приложение, нажмите Ctrl+C в терминале или выполните команду:

```
docker compose down -v
```

Флаг `-v`` также удалит именованный том с данными БД.

В результате выполнения этого упражнения у вас будет полностью контейнеризованное приложение. Все компоненты (фронтенд, бэкенд, БД) работают в своих изолированных средах со своими зависимостями. Любой разработчик может получить проект из репозитория и запустить его одной командой, не тратя время на настройку окружения. Вы можете легко менять версии сервисов, масштабировать их и тестировать взаимодействие между ними. Полученную систему можно развернуть на любом сервере или облачной платформе, поддерживающей Docker.

Контрольные вопросы к упражнению:

1. Зачем в Dockerfile.backend используется базовый образ python:3.11-slim, а не, например, python:3.11? Какие преимущества дает использование slim-образов для конечного контейнера?
2. Объясните, как работает кэширование слоев при сборке образа (docker build)?
3. Как сервисы, описанные в docker-compose.yml (например, frontend, backend, db), находят друг друга?
4. Объясните разницу в результате выполнения команд docker compose down и docker compose rm. Что делает флаг -v в команде docker compose down -v и какие последствия это имеет для ваших данных?
5. Что означает запись depends_on и гарантирует ли она, что сервис-источник уже полностью готов к работе на момент запуска зависимого сервиса?

Упражнение 11. Разработка чат-бота

Цель упражнения: Внедрить интеллектуального ассистента для поддержки пользователей.

Задание:

1. Подготовить базу знаний (БЗ) из агрономических данных.
2. Выбрать и настроить NLP-модель и векторную базу данных.
3. Разработать API-эндпоинт /api/chat в FastAPI для обработки запросов.
4. Реализовать чат-интерфейс на фронтенде.
5. Сформировать отчет с последовательной фиксацией этапов выполнения задания и подтверждением ключевых моментов визуальными снимками экрана (скриншотами).

Ход выполнения упражнения:

1. Подготовка базы знаний

Чат-бот должен быть экспертом. Для этого мы создадим "базу знаний" — набор текстовых документов, на которые модель будет опираться при ответе.

1.1Сбор данных.

Создайте в корне проекта папку knowledge_base/. В ней создайте несколько текстовых файлов с агрономической информацией.

Например:

- fertilizers.txt:

  Аммофос (N:P = 12:52) — комплексное удобрение. Применяется для основного внесения под зерновые, технические и овощные культуры. Обеспечивает растения легкодоступным фосфором на ранних стадиях роста.

- wheat_care.txt:

Яровая пшеница требовательна к влаге. Критические периоды — кущение и выход в трубку. Для борьбы с сорняками применяют гербициды в фазе кущения.

1.2Структура папки проекта:

```
agro-dashboard/
├── backend/
|   └── main.py
├── frontend/
|   └── static/app.js
└── knowledge_base/
    ├── fertilizers.txt
    └── wheat_care.txt
```

2. Настройка бэкенда

Нам понадобятся новые библиотеки: langchain (для организации RAG), faiss-cpu (для быстрого поиска по векторам) и sentence-transformers (для создания этих векторов).

2.1Установите зависимости:

```
pip install langchain faiss-cpu sentence-transformers
```

2.2 Обновите backend/main.py.

- При запуске FastAPI читает все .txt файлы из knowledge_base/.
- С помощью модели all-MiniLM-L6-v2 каждый текст превращается в набор чисел (вектор), который отражает его смысл.
- Эти векторы сохраняются в специальной базе данных FAISS.
- Когда пользователь отправляет сообщение (например, "Какие удобрения нужны?"), мы превращаем его в вектор и ищем в базе FAISS наиболее похожие векторы — то есть, самые релевантные документы.
- Мы берем текст этих документов и возвращаем как ответ.

```
# backend/main.py

from fastapi import FastAPI, HTTPException, Request
from pydantic import BaseModel
import os
import glob
# --- Импорты для RAG ---
from langchain.embeddings import SentenceTransformerEmbeddings
```

```
from langchain.vectorstores import FAISS
from langchain.schema import Document

app = FastAPI()

# --- 1. Загрузка модели и создание базы знаний (выполняется 1 раз при старте)
---
# Выбираем модель для создания векторных представлений текста (embeddings)
embeddings = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")

# Читаем все файлы из папки knowledge_base/
documents = []
knowledge_dir = "../knowledge_base"
for file in glob.glob(os.path.join(knowledge_dir, "*.txt")):
    with open(file, 'r', encoding='utf-8') as f:
        text = f.read()
        # Создаем документ с содержимым файла и его названием в качестве
источника
        documents.append(Document(page_content=text, metadata={"source":
os.path.basename(file)}))

# Создаем векторную базу данных (FAISS) из наших документов
# Это позволяет искать наиболее релевантные фрагменты текста по смыслу
db = FAISS.from_documents(documents, embeddings)


# --- 2. Определение схемы запроса/ответа ---
class ChatRequest(BaseModel):
    message: str

class ChatResponse(BaseModel):
    reply: str

# --- 3. Реализация API-эндпоинта ---
@app.post("/api/chat", response_model=ChatResponse)
async def chat(request: ChatRequest):
    """
    Эндпоинт для общения с чат-ботом.
    """
    user_message = request.message

    # Поиск 2-х наиболее релевантных документов из нашей базы знаний
    docs = db.similarity_search(user_message, k=2)

    # Формируем контекст для ответа из найденных фрагментов
    context = "\n".join([f"Из файла {doc.metadata['source']}:
{doc.page_content}" for doc in docs])

    # Если ничего не найдено, даем общий ответ
    if not context:
        reply = "К сожалению, у меня нет информации по этому вопросу."
    else:
        # В реальном проекте здесь бы работала большая языковая модель (LLM),
        # которая сгенерирует ответ на основе `user_message` и `context`.
        # Для учебного примера мы просто вернем найденный контекст.
        reply = f"Найденная информация:\n{context}"

    return {"reply": reply}
```

3. Реализация фронтенда (Чат-интерфейс)

Обновим frontend/index.html, добавив блок для чата.

```
<!-- frontend/index.html -->
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Агро Чат-бот</title>
    <style>
        body { font-family: sans-serif; max-width: 800px; margin: auto; padding:
20px; }
        #chat-box { border: 1px solid #ccc; padding: 15px; height: 400px;
overflow-y: auto; margin-bottom: 10px; }
        .message { margin: 10px 0; }
        .user { color: #0066cc; font-weight: bold; }
        .bot { color: #800080; font-style: italic; }
    </style>
</head>
<body>
    <h1>Агрономический Чат-бот</h1>
    <div id="chat-box"></div>

    <form id="chat-form">
        <input type="text" id="user-input" placeholder="Введите ваш вопрос..."
autocomplete="off" required>
        <button type="submit">Отправить</button>
    </form>

    <script src="static/app.js"></script>
</body>
</html>
```

Теперь обновим frontend/static/app.js, добавив логику для чата.

```
// frontend/static/app.js

const API_URL = 'http://backend:8000'; // Обращаемся к сервису бэкенда по имени

const chatForm = document.getElementById('chat-form');
const userInput = document.getElementById('user-input');
const chatBox = document.getElementById('chat-box');

// Функция для добавления сообщения в чат
function addMessage(text, sender) {
    const messageDiv = document.createElement('div');
    messageDiv.classList.add('message', sender);
    messageDiv.textContent = text;
    chatBox.appendChild(messageDiv);
    chatBox.scrollTop = chatBox.scrollHeight; // Прокрутка вниз
}
chatForm.addEventListener('submit', async function(event) {
    event.preventDefault();

    const message = userInput.value.trim();
    if (!message) return;
    // Очищаем поле ввода и добавляем сообщение пользователя
    userInput.value = '';
    addMessage(message, 'user');

    try {
        const response = await fetch(`${API_URL}/api/chat`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ message })
        });
```

```
        if (!response.ok) {
            throw new Error('Ошибка соединения с сервером');
        }
        const data = await response.json();
        addMessage(data.reply, 'bot');
    } catch (error) {
        console.error("Ошибка:", error);
        addMessage("Упс, что-то пошло не так. Попробуйте еще раз.", 'bot');
    }
});
\
```

1. При отправке формы сообщение отправляется на эндпоинт /api/chat.
2. Бэкенд обрабатывает запрос и возвращает ответ.
3. Фронтенд получает ответ и добавляет его в окно чата с пометкой "бот".
4. Интерфейс очищается и готов к новому вопросу.

После выполнения всех шагов у вас будет работающий интеллектуальный чат-бот в приложении, которое функционирует примерно таким образом:

1. Пользователь задает вопрос на веб-странице.
2. Фронтенд отправляет вопрос на бэкенд.
3. Бэкенд использует векторную базу данных для поиска релевантной информации в вашей базе знаний.
4. На основе найденной информации формируется ответ, который возвращается пользователю.
5. Вы можете легко расширять знания бота, просто добавляя новые .txt файлы в папку knowledge_base/ без изменения кода.

Контрольные вопросы к упражнению:

1. Опишите ключевое различие между чат-ботом, работающим на основе простой генерации (без доступа к внешним данным), и чат-ботом, использующим подход RAG. Почему для задачи агрономического консультирования подход RAG является более предпочтительным?
2. Какую конкретную задачу решает библиотека FAISS в архитектуре нашего чат-бота?

3. Что такое "встраивание" (embedding) текста в контексте нашего чат-бота? Какую роль в этом процессе играет модель SentenceTransformer? Почему мы преобразуем и запрос пользователя, и документы из базы знаний в векторы перед поиском?
4. Почему загрузка модели SentenceTransformer и создание индекса FAISS (FAISS.from_documents) вынесены за пределы функции-обработчика эндпоинта /api/chat?
5. В текущем коде бэкенда отсутствует компонент "генерации" (LLM). Мы просто возвращаем найденный контекст. Как изменится логика эндпоинта /api/chat, если мы захотим, чтобы бот не просто показывал релевантные документы, а генерировал на их основе связный ответ на языке пользователя? Какой компонент для этого потребуется добавить?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Сысолетин, Е. Г. Разработка интернет-приложений : учебник для вузов / Е. Г. Сысолетин, С. Д. Ростунцев ; под научной редакцией Л. Г. Доросинского. — Москва : Издательство Юрайт, 2025. — 80 с. — (Высшее образование). — ISBN 978-5-534-17124-2. — Текст : электронный // Образовательная платформа Юрайт [сайт]. — URL: **https://urait.ru/bcode/562916**

2. Диков, А. В. Web-программирование на JavaScript : учебное пособие для вузов / А. В. Диков. — Санкт-Петербург : Лань, 2026. — 436 с. — ISBN 978-5-507-54625-1. — Текст : электронный // Лань : электронно-библиотечная система. — URL: **https://e.lanbook.com/book/518225**

3. Диков, А. В. Клиентские технологии веб-дизайна. HTML5 и CSS3 / А. В. Диков. — 2-е изд., стер. — Санкт-Петербург : Лань, 2023. — 188 с. — ISBN 978-5-507-46740-2. — Текст : электронный // Лань : электронно-библиотечная система. — URL: **https://e.lanbook.com/book/318443**

4. Полуэктова, Н. Р. Разработка веб-приложений : учебник для среднего профессионального образования / Н. Р. Полуэктова. — 2-е изд. — Москва : Издательство Юрайт, 2026. — 170 с. — (Профессиональное образование). — ISBN 978-5-534-22183-1. — Текст : электронный // Образовательная платформа Юрайт [сайт]. — URL: **https://urait.ru/bcode/600876**.

5. Флэнаган, Д. JavaScript. Полное руководство : перевод с английского / Д. Флэнаган.— 7-е изд. — Санкт-Петербург : Диалектика, 2021.— 720 с.

Учебное издание

Ермолаева Ольга Сергеевна
Пчелинцева Светлана Вячеславовна

Разработка веб-приложений с интеллектуальными компонентами

Учебно-методическое пособие

Издано в авторской редакции
Корректура автора