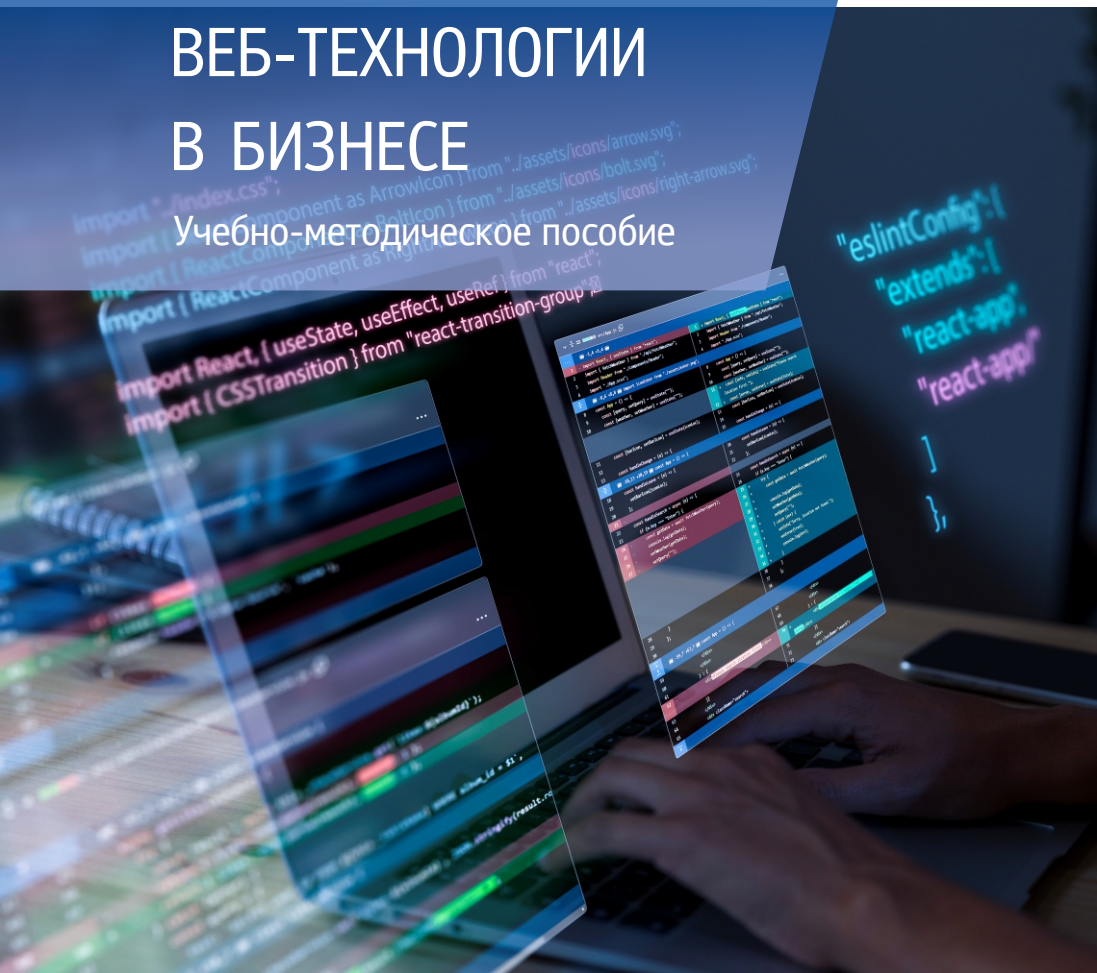
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина
Институт радиоэлектроники и информационных технологий — РТФ

М. А. МЕДВЕДЕВ
М. А. МЕДВЕДЕВА

ВЕБ-ТЕХНОЛОГИИ В БИЗНЕСЕ

Учебно-методическое пособие



Министерство науки и высшего образования
Российской Федерации
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

М. А. Медведев, М. А. Медведева

ВЕБ-ТЕХНОЛОГИИ В БИЗНЕСЕ

Учебно-методическое пособие

Рекомендовано методическим советом
Уральского федерального университета
для студентов вуза, обучающихся по направлению подготовки
09.04.03 — Прикладная информатика

Екатеринбург
Издательство Уральского университета
2024

УДК 004:334.722(075.8)
ББК 16.263.4я73+65.29я73
М42

Рецензенты:

В. М. Кормышев, доцент, кандидат технических наук, завкафедрой информационных технологий и статистики Уральского государственного экономического университета;

Д. Г. Ермаков, доцент, кандидат физико-математических наук, старший научный сотрудник лаборатории научно-информационных ресурсов отдела дифференциальных уравнений Института математики и механики им. Н. Н. Красовского УрО РАН

Научный редактор — д-р физ.-мат. наук, проф. *Д. Б. Берг*

Иллюстрация на обложку взята с сайта https://ru.freepik.com/

Медведев, Максим Александрович.

М42 Веб-технологии в бизнесе : учебно-методическое пособие / М. А. Медведев, М. А. Медведева ; Министерство науки и высшего образования РФ, Уральский федеральный университет. — Екатеринбург : Изд-во Урал. ун-та, 2024. — 168 с. — ISBN 978-5-7996-3905-1. — Текст : непосредственный.

ISBN 978-5-7996-3905-1

В учебно-методическом пособии рассматриваются базовые знания о различных аспектах разработки веб-приложений. Изучаются основные инструменты веб-разработки: язык разметки HTML, каскадные таблицы стилей — CSS, система контроля версий *Git*, *Bootstrap Framework*, язык программирования *JavaScript*, библиотека *React.js*.

Предназначено для студентов и работников, специализирующихся в области прикладной информатики, компьютерных наук и занимающихся разработкой веб-приложений для бизнеса.

УДК 004:334.722(075.8)
ББК 16.263.4я73+65.29я73

ISBN 978-5-7996-3905-1

Оглавление

Введение

Веб-технологии стали неотъемлемой частью современной корпоративной культуры, и их умелое использование сегодня является ключевым фактором успеха бизнеса. Современные веб-технологии помогают перестраивать бизнес-процессы, повышают эффективность и улучшают конкурентоспособность компаний в любой сфере экономики.

Заказчикам ИТ-проектов, которые чаще всего не являются IT-специалистами, важно понимать некоторые аспекты веб-дизайна и веб-разработки для того, чтобы иметь возможность провести обсуждение технических моментов создания проекта с группами разработчиков или поставщиками ПО. Знание веб-технологий дает заказчикам концептуальное представление о том, как будет построен проект с технической точки зрения.

Важной фигурой в этом цифровом ландшафте является веб-разработчик. От создания интерактивных веб-сайтов до внедрения сложных систем управления данными — веб-разработчики играют важную роль в преобразовании бизнеса.

Веб-разработчик — одна из современных перспективных и карьерных профессий в IT-отрасли. Технологии веб-разработки в руках специалиста позволяют создавать веб-сайты и приложения, которые помогают решать многообразие задач в сфере бизнеса и маркетинга.

Для того чтобы создавать веб-приложения или веб-сайты, необходимо знать минимальный набор веб-технологий, без которых веб-разработка невозможна. Любой веб-разработчик должен иметь опыт работы со следующими инструментами: HTML,

CSS, *JavaScript*, одним из фреймворков для разработки интерфейсов (например, *React.js*, *Angular.js* или *Vue.js*), языком запросов SQL для работы с базами данных, одним из серверных языков программирования, а также IDE (интегрированная среда разработки), в которой используется этот язык. Конечно, это далеко не полный список. Однако эти технологии обязательны для изучения хотя бы на базовом уровне.

Ссылки на полезные ресурсы по изучению веб-технологий можно найти внутри глав данного пособия, а также в списке рекомендуемой литературы по соответствующей теме.

1. Основные этапы веб-разработки

В современную цифровую эпоху веб-технологии стали неотъемлемой частью нашей жизни, позволяя предприятиям, организациям и частным лицам обеспечить свое присутствие в Сети. Понимание этапов веб-разработки имеет решающее значение для всех, кто заинтересован в создании веб-сайта, приложения или в продолжении карьеры в области IT.

Далее, рассмотрим основные этапы веб-разработки проектов и изучим их содержание.

1.1. Этап планирования

Первым этапом веб-разработки является стадия планирования. На этом этапе важно определить назначение, задачи и целевую аудиторию проекта. Ключевые мероприятия на этом этапе включают:

- определение целей создания проекта и желаемых результатов;
- проведение маркетинговых исследований и анализа конкурентов;
- определение целевой аудитории и ее потребностей;
- разработку карты веб-сайта или структуры для описания страниц веб-сайта и навигации по ним;
- создание графика и бюджета для проекта.

Эти мероприятия закладывают прочную основу для успешного проекта веб-разработки, четко определяя цели и приводя их в соответствие с потребностями и ожиданиями целевой аудитории.

1.2. Этап разработки технического задания

Этот этап предполагает создание всеобъемлющего документа с техническим заданием, в котором излагаются конкретные требования, функциональные возможности и технические детали проекта. Техническое задание служит основой для команды разработчиков и гарантирует, что все участники, вовлеченные в проект, согласовали между собой все задачи и понимают масштаб проекта. В ключевые аспекты этапа разработки технического задания входят:

- *сбор требований* — сотрудничество с заинтересованными сторонами, клиентами и командой разработчиков для сбора всех необходимых требований к проекту, понимание желаемых характеристик, функциональности и любых конкретных технических ограничений или предпочтений;
- *определение технических спецификаций* — основываясь на требованиях, этап технического задания фокусируется на уточнении технических деталей проекта. Здесь происходит выбор подходящих языков программирования, фреймворков и инструментов, которые будут использоваться при разработке проекта, определение структуры базы данных, описание мер безопасности и определение требований к интеграции с другими системами, если это применимо;
- *создание прототипов* — в рамках разработки технического задания могут быть созданы прототипы с низкой точностью, чтобы визуализировать макет и UX/UI проекта. Прототипы помогают подтвердить правильность технического подхода и собрать отзывы заинтересованных сторон;
- *документирование технических спецификаций* — документ с техническим заданием должен содержать подробную информацию об архитектуре проекта, структурах данных, ролях пользователей и разрешениях, сторонних интеграциях, API-интерфейсах и любых других техниче-

ских аспектах. В нем также должны быть указаны любые конкретные требования к производительности или масштабируемости;

– *рассмотрение и утверждение* — документ технического задания должен быть рассмотрен заинтересованными сторонами проекта, включая клиентов, проектировщиков, разработчиков и персонал по обеспечению качества, чтобы убедиться, что он точно отражает требования проекта. Любые необходимые изменения или уточнения должны быть внесены до перехода к этапам проектирования и разработки.

Этап разработки технического задания имеет решающее значение для установления четкого понимания технических аспектов проекта и содействия эффективной коммуникации и сотрудничеству между командой разработчиков. Он служит ориентиром на всех последующих этапах проектирования, разработки и тестирования, гарантируя соответствие поставленным целям и техническим требованиям.

1.3. Этап проектирования

Как только техническое задание создано, мы переходим к стадии проектирования. На этом этапе основное внимание уделяется визуальным аспектам и пользовательскому интерфейсу проекта. Этап проектирования включает:

– создание прототипов или низкокачественных макетов для визуализации структуры проекта;
– разработку элементов пользовательского интерфейса (UI), таких как кнопки, формы, меню и т. п.;
– выбор подходящих цветовых схем, типографики и изображений;
– обеспечение адаптивности дизайна проекта и его совместимости с различными устройствами и размерами экранов.

1.4. Этап разработки

На стадии разработки проект начинает обретать форму. Здесь происходит превращение дизайнерских концепций в полнофункциональный продукт. Этап разработки включает:

- написание HTML, CSS и JavaScript-кода для создания структуры, стилей и интерактивности веб-страниц;
- интеграция с системой управления контентом (CMS) или платформой электронной коммерции, если требуется;
- внедрение систем баз данных для вывода динамического контента и хранения данных;
- оптимизация производительности проекта, включая скорость загрузки и отзывчивость;
- проведение тщательного тестирования и отладки для обеспечения функциональности и кроссбраузерной совместимости.

1.5. Этап создания контента

Стадия создания контента часто совпадает со стадией разработки. Она фокусируется на создании привлекательного и релевантного контента для проекта и включает в себя:

- создание текстового контента для каждой веб-страницы;
- создание или выбор подходящих визуальных элементов, таких как изображения, видео и инфографика;
- оптимизация контента для поисковых систем (SEO) путем включения релевантных ключевых слов и метаданных;
- обеспечение доступности контента и соответствия рекомендациям по обеспечению доступности в Интернете (WCAG).

1.6. Этап развертывания

Как только проект разработан и его содержимое готово, наступает время для развертывания. Этап развертывания включает в себя:

- настройку хостинга и регистрацию доменного имени;
- загрузку всех файлов проекта на сервер;
- настройку параметров сервера и обеспечение надлежащих мер безопасности;
- проведение окончательных проверок для того, чтобы убедиться в корректной работе всех функций проекта;
- запуск проекта для публичного доступа.

1.7. Этап поддержки и обновлений

После запуска проекта работа над ним не заканчивается. Регулярное техническое обслуживание и обновления имеют решающее значение для его успеха. Мероприятия на этом этапе включают:

- мониторинг производительности, времени безотказной работы и безопасности;
- внедрение обновлений в используемые CMS, плагины и фреймворки;
- анализ аналитики проекта для получения информации о поведении пользователей и внесение улучшений;
- добавление нового контента, функций или разделов по мере необходимости;
- создание резервных копий и проверка безопасности для защиты данных проекта.

Веб-разработка включает в себя несколько этапов, каждый из которых имеет уникальное содержание и цели. Понимая этапы и их содержание, вы будете лучше подготовлены к планированию, дизайну, разработке и сопровождению успешного проекта. Необходимо помнить, что веб-разработка — это непрерывный и постоянно изменяющийся в техническом плане

процесс. Для создания привлекательных и удобных в использовании веб-сайтов и приложений необходимо быть в курсе новейших технологий и тенденций.

Вопросы для самопроверки

1. Какие основные этапы веб-разработки Вы можете выделить?
2. Что включает в себя этап планирования веб-разработки?
3. Какие этапы включаются в разработку дизайна веб-сайта или приложения?
4. Что такое веб-дизайн, как он отличается от веб-разработки?
5. Какие шаги включает в себя этап запуска и развертывания веб-проекта?
6. Почему поддержка и обновление веб-сайта являются важной частью веб-разработки?

2. Редактор кода

Редакторы кода — это программы, предназначенные для создания и редактирования исходного кода компьютерных программ. Они предоставляют широкий набор функций и инструментов для облегчения работы разработчика. Ниже представлены основные функции редакторов кода:

1) *подсветка синтаксиса* — редакторы автоматически подсвечивают элементы кода (ключевые слова, переменные, комментарии и т. п.) разными цветами, чтобы облегчить его чтение и понимание;
2) *автодополнение* — эта функция предлагает подсказки и автодополнение при вводе кода. Она ускоряет процесс написания кода и уменьшает количество ошибок;
3) *отступы и форматирование* — редакторы позволяют легко управлять отступами и форматированием кода, что делает его более читаемым и структурированным;
4) *быстрый доступ к функциям* — часто используемые функции и методы можно быстро вызвать с помощью горячих клавиш или команд;
5) *поиск и замена* — редакторы предоставляют мощные инструменты для поиска и замены текста в коде. Это позволяет быстро находить и исправлять ошибки или изменять части кода;
6) *множество языков программирования* — хорошие редакторы кода поддерживают множество языков программирования, что позволяет разработчикам работать с разнообразными технологиями;

7) *интеграция с системами контроля версий* — редакторы обычно интегрированы с системами контроля версий, такими как *Git*, что упрощает отслеживание изменений и совместную работу над кодом;
8) *поддержка плагинов и расширений* — многие редакторы кода могут быть расширены с помощью плагинов, добавляющих новые функции и возможности;
9) *просмотр исходных файлов* — возможность быстро просматривать исходные файлы и переходить между ними упрощает навигацию в больших проектах;
10) *интегрированный терминал* — некоторые редакторы предоставляют встроенный терминал, который позволяет выполнять команды и скрипты без выхода из текущего проекта;
11) *режим разработки и отладки* — некоторые редакторы предоставляют интегрированные средства для отладки кода, что упрощает выявление и устранение ошибок.

Для написания кода проекта можно использовать различные редакторы кода, например, *NotePad++*, *SubLime Text*, *MS Visual Studio Code* (*VS Code*). Далее, мы будем использовать последний из перечисленных — *VS Code*.

VS Code — это бесплатный редактор от компании *Microsoft*. Он достаточно легковесный, обладает высокой скоростью работы, прост в установке. В редакторе имеется возможность установки различных плагинов для ускорения работы. Ниже представлены основные особенности *VS Code*.

1. *Редактор кода с подсветкой синтаксиса*. *VS Code* предоставляет возможность автодополнения и подсветки синтаксиса для большого числа языков программирования.
2. *Интеграция с Git*. Отличная интеграция с системой контроля версий *Git*, что упрощает отслеживание изменений в коде и работу в команде.
3. *Расширения и плагины*. Можно легко расширять функциональность *VS Code* с помощью множества доступных расширений и плагинов, которые позволяют до-

бавлять поддержку для разных языков, фреймворков и инструментов.

4. *Интегрированный отладчик*. Встроенный отладчик, который поддерживает множество языков программирования, позволяет разработчикам отслеживать и исправлять ошибки в коде.
5. *Автоматическое завершение кода*. Функция автодополнения (*IntelliSense*) для более быстрого и точного написания кода.
6. *Управление зависимостями*. Встроенные инструменты для управления зависимостями проекта и установки сторонних библиотек.
7. *Интегрированный терминал*. Возможность использовать командный терминал внутри *VS Code* для выполнения команд и скриптов.
8. *Работа с расширенными языками разметки*. Поддержка HTML, CSS, и *JavaScript*, а также языков разметки и стилей, делает *VS Code* полезным для веб-разработчиков.
9. *Система автоматической проверки кода*. Возможность выполнения статического анализа кода и автоматической проверки на наличие ошибок.
10. *Интеграция с Docker*. Возможность управления контейнерами *Docker* непосредственно из *VS Code*.
11. *Работа с Jupyter Notebooks*. Поддержка интерактивных *Jupyter Notebooks* для анализа данных и машинного обучения.
12. *Система сниппетов*. Возможность создавать и использовать собственные сниппеты кода для повышения производительности.
13. *Инструменты для работы с расширенными текстовыми файлами*. Возможность работы с большими текстовыми файлами и логами.
14. *Темы оформления и настройки*. Пользователи могут настраивать внешний вид рабочей области, выбирая из различных тем оформления и настроек.

15. *Интеграция с облачными сервисами*. Возможность интеграции с различными облачными сервисами, такими как *Azure* и AWS.
16. *Легкая установка и обновление*. Установка и обновление расширений и плагинов выполняются в несколько кликов.

Установка редактора кода

Для установки *VS Code* необходимо:
- перейти на официальный веб-сайт[1] редактора;
- скачать дистрибутив;
- установить редактор на компьютер, следуя инструкциям установщика.

После установки редактора кода его необходимо настроить для дальнейшей работы. *VS Code* дает возможность установить множество различных плагинов, которые помогают ускорить написание кода и автоматизировать множество рутинных задач, а также писать код на различных языках программирования.

Довольно часто в *VS Code* используются следующие плагины:
- *Auto Close Tag*;
- *Auto Complete Tag*;
- *Auto Rename Tag*;
- *ES7+ React/Redux/React Native/JS snippets*;
- *C# for Visual Studio*;
- *HTML CSS Support*;
- *Intellisense for CSS class names in HTML*;
- *Live Server*.

Для синхронного обновления веб-страницы в браузере используется плагин *Live Server*.

Чтобы использовать этот плагин, необходимо сделать следующее:

1) в редакторе *VS Code* переходим на вкладку «Расширения» (*Extansions*). В строке поиска вводим название

[1] Visual Studio Code : [website]. URL: https://code.visualstudio.com/ (date of accessed: 02.02.2024).

необходимого плагина и нажимаем кнопку «Установить» (*Install*);

2) для запуска плагина нажимаем на иконку (*Go Live*), которая находится на нижней панели редактора (с правой стороны).

VS Code запустит встроенный веб-сервер и откроет html-страницу в браузере по адресу: http://127.0.0.1:5500/index.html.

Если в код страницы будут внесены какие-либо правки, и она будет сохранена, то все изменения будут отображаться в браузере автоматически.

Многие редакторы кода поддерживают возможность командной работы посредством использования системы управления версиями *Git*.

Вопросы для самопроверки

1. Что такое редакторы кода, и какую роль они играют в веб-разработке?
2. Какие основные функции предоставляют редакторы кода для улучшения процесса написания кода?
3. Какие функции редактора кода помогают автоматизировать и упростить написание кода?
4. Какие инструменты и плагины можно установить в редактор кода для улучшения его функциональности?
5. Какие редакторы кода с открытым исходным кодом популярны среди веб-разработчиков?
6. Какие редакторы кода подходят для разработки веб-приложений на разных языках программирования (например, *JavaScript*, *Python*, *Ruby*)?

3. Система контроля версий Git

Крайне важным инструментом для работы программиста в команде является система контроля версий.

Системы контроля версий — это инструменты, используемые для отслеживания изменений в файлах и проектах с течением времени и управления ими. Они обеспечивают совместную работу, предоставляют возможности резервного копирования и восстановления, а также облегчают проведение экспериментов.

3.1. Что такое Git?

Git — это распределенная система контроля версий, предназначенная для быстрой и эффективной работы как с небольшими, так и с крупными проектами. Она была создана Линусом Торвальдсом (*Linus Torvalds*) в 2005 г. для управления разработкой ядра *Linux*.

Независимо от того, являетесь ли вы разработчиком, дизайнером или создателем контента, *Git* может помочь в отслеживании изменений, сотрудничать с другими пользователями и вести историю вашей работы.

Преимущества *Git*:

1) *распределенный характер* — у каждого пользователя есть полная копия проекта, что позволяет работать в автономном режиме и легко создавать ветки проекта;
2) *скорость и производительность* — *Git* разработан таким образом, чтобы работать с высокой производительностью даже с большими репозиториями;

3) *целостность данных* — *Git* использует криптографическую хэш-функцию для обеспечения целостности файлов;
4) *ветвление и слияние* — *Git* предоставляет мощные возможности ветвления и слияния проектов, обеспечивая параллельную разработку и легкую совместную работу;
5) *отслеживание изменений* — *Git* ведет полную историю всех изменений, внесенных в проект, позволяя отслеживать авторов изменений в коде. Из истории можно легко понять, кто и с какой целью внес изменения.

Git позволяет отслеживать изменения в любом наборе файлов. Обычно *Git* используется для координации работы программистов, которые совместно разрабатывают исходный код во время создания программного обеспечения.

Для начала работы прежде всего необходимо установить *Git* на локальный компьютер, а также создать аккаунт на любом веб-сервисе управления репозиториями (*GitHub*, *GitLab* и т. п.).

GitHub — это сайт онлайн-хранилище для истории версий проекта (удаленный репозиторий). Его работа основана на *Git*. *GitHub* позволяет размещать на удаленном репозитории код проекта для дальнейшей совместной работы.

Для регистрации аккаунта на *GitHub* можно воспользоваться ссылкой: https://github.com/.

3.2. Установка Git

Чтобы установить *Git* на свой компьютер, выполните следующие действия.

1. ***Windows***:
   - загрузите установщик *Git* с официального веб-сайта: https://git-scm.com/downloads;
   - запустите загруженный установщик и следуйте инструкциям мастера установки;
   - выберите желаемые параметры установки и завершите процесс установки.

2. ***MacOS***:
   - *Git* обычно предустановлен на *MacOS*. Откройте терминал и введите команду “git-version”, чтобы проверить, установлен ли *Git*;
   - если он не установлен, вы можете загрузить установщик для *MacOS* по ссылке: https://git-scm.com/downloads;
   - запустите загруженный установщик и следуйте инструкциям мастера установки.
3. ***Linux***:
   - *Git*, возможно, уже доступен в менеджере пакетов вашего дистрибутива *Linux*. Вы можете использовать менеджер пакетов для установки *Git*. Например, в ОС *Ubuntu* вы можете использовать команду “sudo apt-get install git”;
   - если он недоступен в вашем менеджере пакетов, или вам нужна последняя версия, вы можете загрузить его по ссылке: https://git-scm.com/downloads;
   - следуйте инструкциям, приведенным на веб-сайте, чтобы установить *Git* в ваш дистрибутив *Linux*.

3.3. Настройка Git

Работа с любым инструментом всегда начинается с его настройки. *Git* можно настроить один раз и изменять настройки только по мере необходимости. Все команды, приведенные ниже, запускаются из командной строки локального компьютера.

Установка имени пользователя, от которого будут отправлены изменения:

git config -- global user.name “User Name”

Установка адреса электронной почты. Пожалуйста, обратите внимание, что адрес электронной почты должен совпадать с тем, который был использован при регистрации учетной записи *GitHub*:

git config -- global user.email “mail@gmail.com”

Установка текстового редактора, который будет открывать файлы для разрешения конфликтов:

git config -- global core.editor “editor”

С помощью команды:

git config -- list

вы можете увидеть список всех сделанных настроек.

3.4. Клонирование репозитория

Допустим, руководитель вашей команды поручил вам новый проект. Этот проект находится на удаленном репозитории (в нашем случае это будет *GitHub*). Вам нужно скопировать этот репозиторий, внести в него некоторые изменения и загрузить обратно в облако. Давайте посмотрим, как это можно сделать.

Чтобы клонировать репозиторий, введите команду:

git clone “адрес_репозитория”

Репозиторий клонируется в текущую папку, отрытую из командной строки.

3.5. Работа с изменениями

Любая работа с изменениями начинается с получения последней версии удаленного репозитория. Вы можете получить последнюю версию с помощью команды:

git pull [<options>] [<repository> [<refspec>...]]

Будьте осторожны: вызов этой команды приведет к удалению всех незафиксированных изменений в локальном репозитории.

После внесения любых изменений в проект вы можете просмотреть состояние файлов с помощью команды:

git status

В командной строке будут показаны файлы, которые были изменены, удалены, а также будут указаны имена новых файлов, которые необходимо добавить для отслеживания изменений.

Чтобы добавить отслеживание изменений для новых файлов, необходимо использовать команду:

git add <filename> <filename>

чтобы добавить несколько файлов по имени.

Команда:

git add .

добавляет отслеживание для всех новых файлов из текущего каталога.

Команда:

git add -A

добавляет удаленные файлы не только из текущего каталога, но и из всего локального хранилища.

Файлы также можно удалить, для этого существует команда:

git rm <filename> <filename>

которая удаляет файлы по их названию.

После того как все новые файлы добавлены, а старые удалены, вы можете зафиксировать изменения. Фиксация изменений (команда “git commit”) очень важна, потому что до тех пор, пока эта команда не будет выполнена, ваши локальные изменения нигде не будут записаны. Чтобы зафиксировать изменения, необходимо ввести команду:

git commit -m “comment to commit”

Стоит отметить, что необходимо правильно разбивать изменения на коммиты и давать полные комментарии к ним.

Если вы внесли изменения и хотите быстро отменить их, то воспользуйтесь командой:

git reset

которая отменяет все незафиксированные изменения.

По умолчанию эта команда удаляет только из индекса. И команда:

git reset -- hard

безвозвратно удаляет незафиксированные текущие изменения из локального репозитория и из индекса.

Поскольку все вышеперечисленные действия выполняются в локальной копии репозитория, эта копия должна быть отправлена на сервер, чтобы другие участники могли получить последнюю версию проекта. Для этого существует команда:

git push

которая отправляет все зафиксированные изменения в удаленный репозиторий.

3.6. Работа с файлами

С помощью *Git* вы можете создавать ветки и перемещаться по ним.

Команда:

git checkout -b “branch-name”

создаст ветку с указанным именем и автоматически переключится на нее.

После создания ветка может быть отправлена на сервер с помощью команды:

git push origin “branch-name”

Аналогично вы можете перенести ветку из удаленного репозитория на локальный компьютер с помощью команды:

git checkout origin/”branch-name” -b “branch-name”

Чтобы не хранить имена ветвей в памяти и не искать их названия, существуют две специальные команды, которые позволяют просматривать все существующие ветви локального репозитория:

git branch

или все существующие ветви удаленного репозитория:

git branch -r

Вы можете переключиться на любую локальную ветку с помощью команды:

git checkout “branch-name”

После работы над проектом в репозитории могут остаться различные ненужные неотслеживаемые файлы и прочий мусор. Чтобы удалить все ненужные файлы, используйте команду:

git clean -f -d

Ниже представлено несколько дополнительных ресурсов для изучения *Git* и процесса его установки на локальный компьютер.

1. *Официальная документация по Git*. Официальная документация по *Git* является отличным ресурсом для изучения его с нуля. Она охватывает все — от базовых до продвинутых тем, содержит подробные объяснения концепций и команд *Git*. Вы можете найти ее по адресу: https://git-scm.com/doc.
2. *Книга* “*Pro Git*” Скотта Чакона и Бена Страуба, которая подробно описывает *Git*. Она начинается с основ и переходит к более сложным темам, таким как ветвление, слияние и перебазирование. Книга доступна бесплатно онлайн по адресу: https://git-scm.com/book/en/v2.
3. *Учебные пособия по Git на Atlassian Bitbucket*. *Bitbucket* предоставляет серию пошаговых руководств, которые охватывают основы *Git*. Эти учебные пособия удобны для начинающих и содержат практические примеры. Вы можете получить к ним доступ по адресу: https://www.atlassian.com/git/tutorials.
4. *Учебная лаборатория GitHub*. Учебная лаборатория *GitHub* предлагает интерактивные практические курсы по изучению *Git* и *GitHub*. Они обеспечивают управляемый процесс обучения с использованием реальных сценариев и упражнений. Вы можете получить доступ к курсам по адресу: https://lab.github.com/.
5. *GitKraken Git Client*. *GitKraken* — популярный Git-клиент, который предлагает визуально привлекательный и удобный интерфейс. На этом веб-сайте представлены учебные пособия и ресурсы, которые помогут освоить *Git* и начать работу с их клиентом: https://www.gitkraken.com/learn-git.

Вопросы для самопроверки

1. Что такое системы управления версиями (VCS), и какую роль они играют в разработке программного обеспечения?
2. Какие преимущества предоставляет использование систем управления версиями в разработке?
3. Как создать новый репозиторий *Git* для вашего проекта?
4. Какие команды *Git* используются для добавления файлов в индекс (*staging area*) перед коммитом?
5. Что такое коммит (*commit*) в *Git*, и как создать новый коммит с сообщением?
6. Как создать и переключиться на новую ветку (*branch*) в *Git*?
7. Как создать копию удаленного репозитория на вашем локальном компьютере (клонирование)?
8. Как отправить изменения из локального репозитория в удаленный репозиторий (*push*)?
9. Какие команды *Git* используются для отмены изменений и восстановления предыдущих версий файлов?
10. Какие ресурсы и инструменты доступны для изучения *Git* и систем управления версиями?

4. HTML

HTML (*Hyper Text Markup Language*) — это стандартный язык, используемый для создания и структурирования веб-страниц во Всемирной паутине. Он составляет основу каждого веб-сайта, который вы посещаете, позволяя представлять информацию структурированным и интерактивным образом.

HTML предоставляет набор тегов (или элементов), которые определяют структуру и содержимое веб-страницы. Эти теги заключены в угловые скобки (< >) и используются для описания различных частей документа. Используя комбинацию этих тегов, мы можем форматировать текст, добавлять изображения, создавать ссылки, добавлять на страницу видео и многое другое. Веб-браузер интерпретирует эти теги соответствующим образом для отображения веб-страницы.

4.1. Основные теги HTML и их использование

Начнем с рассмотрения некоторых основных тегов HTML и их использования.

1. Тег **<html>**: этот тег используется для определения начала и конца HTML-документа. Все остальные HTML-теги помещаются внутрь данного тега.
2. Тег **<head>**: внутри тега <html> размещается тег <head>, который содержит метаданные о документе, такие как заголовок, кодировка символов, связанные таблицы стилей и скрипты.

3. Тег **<body>**: внутри тега <html>, после тега <head>, размещается тег <body>, который представляет основное содержимое веб-страницы. Все, что мы хотим отобразить на странице, находится внутри этого тега.
4. Теги от **<h1>** до **<h6>**: эти теги используются для заголовков разного размера, причем <h1> является самым большим, а <h6> — самым маленьким.
5. Тег **<p>**: этот тег используется для определения абзацев. Текст, заключенный в теги <p>, будет отображаться в виде отдельного блока текста.

Некоторые распространенные варианты использования HTML:

1) *создание персонального веб-сайта*. HTML обеспечивает основу для создания персональных веб-сайтов, на которых вы можете демонстрировать свои навыки, портфолио или вести блог. Используя HTML-теги, вы можете структурировать контент и добавлять различные элементы, чтобы сделать ваш веб-сайт визуально привлекательным;
2) *разработка онлайн-форм*. HTML позволяет создавать формы, которые собирают вводимые пользователем данные, такие как: контактная информация или ответы на опрос. Элементы формы, такие как текстовые поля, флажки, переключатели и кнопки для отправки данных, можно использовать для создания интерактивных форм;
3) *разработка веб-сайтов электронной коммерции*. HTML играет решающую роль в разработке интернет-магазинов. Вы можете создавать списки товаров, корзины покупок и процессы оформления заказа, используя HTML-теги в сочетании с другими веб-технологиями, такими как CSS (каскадные таблицы стилей) и *JavaScript*;
4) *встраивание мультимедиа*. HTML позволяет встраивать мультимедийные элементы, такие как изображения, видео и аудиофайлы, в веб-страницы. Таким образом вы можете обогатить пользовательский опыт за счет плавного включения визуального и интерактивного контента.

Чтобы узнать больше о HTML и его различных тегах, атрибутах, существует несколько информативных онлайн-ресурсов:

- *MDN Web Docs*: HTML: https://developer.mozilla.org/en-US/docs/Web/HTML;
- *W3Schools HTML Tutorial*: https://www.w3schools.com/html/;
- *HTML Dog: The Best-Practice Guide to XHTML and CSS*: http://www.htmldog.com/.

Далее, рассмотрим некоторые основные HTML-теги более подробно.

<html>: этот тег служит корневым элементом HTML-документа и содержит в себе все остальные HTML-теги. Он указывает на то, что содержимое внутри файла является HTML-документом.

<head>: этот тег содержит метаданные о документе, такие как заголовок, кодировка символов, связанные таблицы стилей и скрипты. Он не отображается на самой веб-странице, но предоставляет важную информацию браузеру и поисковым системам.

<body>: этот тег представляет основное содержимое веб-страницы. Он содержит все видимые элементы, которые пользователи увидят при посещении страницы. Текст, изображения, ссылки, заголовки, абзацы и другие HTML-элементы размещаются внутри тега <body>.

<h1> — **<h6>**: эти теги используются для создания заголовков разного размера, где <h1> представляет самый большой заголовок, а <h6> — самый маленький. Например:

```
***
<h1>This is the largest heading</h1>
<h2>This is a subheading</h2>
<h3>Another subheading</h3>
                                                    ***
```

<p>: этот тег используется для определения абзацев текста. Он создает новую строку и добавляет интервал до и после содержимого. Например:

```
***
<p>This is a paragraph of text.</p>
                                                    ***
```

<a>: этот тег создает гиперссылку на другую веб-страницу или определенное местоположение на той же странице. Он обычно используется для навигации по меню, перехода на внешние веб-сайты или создания внутренних ссылок внутри документа. Например:

```
***
<a href="https://www.urfu.ru">Visit URFU website</a>
                                                    ***
```

<img>: этот тег используется для встраивания изображений на веб-страницу. Для данного тега требуется определить атрибут “src” для указания источника изображения (URL или путь к файлу) и атрибут “alt” для предоставления альтернативного текста в случае проблем с загрузкой изображения. Например:

```
***
<img src="image.jpg" alt="A beautiful image">
                                                    ***
```

<ul> и **<li>**: эти теги используются для создания неупорядоченных (маркированных) списков. Тег **<ul>** представляет весь список, и каждый элемент списка определяется с помощью тега **<li>**. Например:

```
***
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
                                                    ***
```

<ol> и **<li>**: эти теги используются для создания упорядоченных (нумерованных) списков. Тег **<ol>**, подобно тегу **<ul>**, представляет список, и каждый элемент определяется с помо-

щью тега **<li>**. Разница в том, что упорядоченные списки пронумерованы. Например:

```
***
<ol>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ol>
                                                    ***
```

** **: этот тег является универсальным контейнером, используемым для группировки и стилизации содержимого. Он часто используется для верстки или применения стилей CSS к определенному разделу веб-страницы. Например:

```
***
<div>
  <h2>Section Title</h2>
  <p>This is the content of the section.</p>
</div>
                                                    ***
```

Выше мы рассмотрели только несколько примеров основных HTML-тегов и их использование. HTML предоставляет широкий спектр тегов и атрибутов для структурирования и форматирования содержимого веб-страниц. Творчески комбинируя и размещая эти теги, вы можете создавать сложные и визуально привлекательные веб-страницы.

4.2. Семантическая верстка

Семантическая верстка в HTML относится к практике использования HTML-тегов, которые передают смысл и описывают структуру контента, а не просто внешний вид. Используя семантические теги, вы делаете свой HTML-код более осмысленным, доступным и дружественным по отношению к поисковым системам.

Рассмотрим несколько примеров семантической компоновки документа:

<header>: тег <header> представляет вводный контент или верхнюю часть веб-страницы (шапку). Обычно он включает в себя логотип веб-сайта, навигационное меню или заголовок страницы. Например:

```
***
<header>
  <h1>My Website</h1>
  <nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>
</header>
                                                    ***
```

<nav>: тег <nav> используется для определения раздела навигационных ссылок. Он представляет собой набор ссылок, которые позволяют пользователям перемещаться по веб-сайту или различным разделам веб-страницы. Например:

```
***
<nav>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</nav>
                                                    ***
```

<main>: тег <main> представляет основную область содержимого веб-страницы. Обычно он содержит основной контент, уникальный для данной веб-страницы. Его следует использовать на веб-странице только один раз. Например:

```
***
<main>
  <h1>Welcome to My Website</h1>
  <p>Here is some important information.</p>
</main>
                                                    ***
```

<article>: тег <article> используется для определения самодостаточной, независимой части контента на веб-странице. Это может быть запись в блоге, новостная статья, сообщение на форуме или любой другой контент, который имеет смысл сам по себе. Например:

```
***
<article>
  <h2>Article Title</h2>
  <p>This is the content of the article.</p>
</article>
                                                    ***
```

<section>: тег <section> используется для разделения содержимого веб-страницы на различные разделы или группы. Это помогает упорядочить связанный контент. Например:

```
***
<section>
  <h2>About Me</h2>
  <p>Some information about me goes here.</p>
</section>
<section>
  <h2>My Skills</h2>
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript</li>
  </ul>
</section>
                                                    ***
```

<aside>: тег <aside> представляет контент, который косвенно связан с основным контентом, но может рассматриваться

отдельно от него. Он часто используется для боковых панелей, цитат или рекламы. Например:

```
***
<aside>
  <h3>Related Articles</h3>
  <ul>
    <li><a href="#">Article 1</a></li>
    <li><a href="#">Article 2</a></li>
    <li><a href="#">Article 3</a></li>
  </ul>
</aside>
                                                    ***
```

Выше мы рассмотрели примеры семантических HTML-тегов, которые помогают структурировать и описывать содержимое веб-страницы. Правильно используя эти теги, вы предоставляете дополнительный контекст, помогающий программам чтения с экрана, поисковым системам и другим устройствам понять назначение и иерархию вашего контента.

Семантический HTML не только улучшает доступность и оптимизацию в поисковых системах, но и делает код более удобным в обслуживании и понятным для других разработчиков.

Ниже представлен пример кода для веб-страницы (*Pizza Order*) с семантическим макетом:

```
***
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">
  <title>Pizza Order</title>
  <style>
    /* Some basic styling for demonstration
purposes */
    body {
      font-family: Arial, sans-serif;
```

```
      margin: 0;
      padding: 20px;
    }
    header {
      background-color: #f2f2f2;
      padding: 10px;
      text-align: center;
    }
    nav {
      margin-bottom: 20px;
    }
    main {
      margin-bottom: 20px;
    }
    section {
      padding: 10px;
      background-color: #f9f9f9;
      margin-bottom: 10px;
    }
    footer {
      background-color: #f2f2f2;
      padding: 10px;
      text-align: center;
    }
  </style>
</head>
<body>
  <header>
    <h1>Pizza Order</h1>
    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">Menu</a></li>
        <li><a href="#">Order</a></li>
        <li><a href="#">Contact</a></li>
      </ul>
    </nav>
  </header>
```

```
  <main>
    <section>
      <h2>Choose Your Pizza</h2>
      <p>Select your favorite pizza from our menu:</p>
      <ul>
        <li>Margherita</li>
        <li>Pepperoni</li>
        <li>Hawaiian</li>
        <li>Veggie Supreme</li>
      </ul>
    </section>

    <section>
      <h2>Customize Your Pizza</h2>
      <p>Customize your pizza with the following
options:</p>
      <ul>
        <li>Size: <select>
          <option>Small</option>
          <option>Medium</option>
          <option>Large</option>
          </select></li>
        <li>Toppings: <input type="checkbox">
Pepperoni <input type="checkbox"> Mushrooms
<input type="checkbox"> Olives</li>
        <li>Crust: <input type="radio" name="crust"
value="thin"> Thin <input type="radio" name="crust"
value="thick"> Thick</li>
      </ul>
    </section>

    <section>
      <h2>Place Your Order</h2>
      <p>Enter your details and submit your order:</p>
      <form>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name"
required>
```

```
        <br>
        <label for="address">Address:</label>
        <textarea id="address" name="address"
required></textarea>
        <br>
        <input type="submit" value="Submit Order">
      </form>
    </section>
  </main>

  <footer>
    &copy; 2023 Pizza Order. All rights reserved.
  </footer>
</body>
</html>
```

***

В этом примере мы структурировали веб-страницу с помощью семантических HTML-тегов. <header> содержит заголовок страницы и навигационное меню. Тег <nav> представляет навигационные ссылки. Основное содержимое заключено в тег <main>, причем каждый раздел содержимого заключен в тег <section>. Нижний колонтитул помечается с помощью тега <footer>.

Представленный CSS (стили страниц) предназначен для демонстрационных целей и может быть дополнительно настроен для достижения желаемого визуального эффекта.

Вопросы для самопроверки

1. Как расшифровывается аббревиатура HTML?
2. Как создать заголовок в HTML-документе? Какие уровни заголовков поддерживаются в HTML-документе?
3. Какие HTML-теги используются для создания списков (ненумерованных и упорядоченных)?

4. Какой HTML-тег используется для вставки гиперссылки на другую веб-страницу?
5. Как создать изображение на веб-странице с помощью HTML-тега?
6. Какой HTML-тег используется для создания абзаца текста?

5. CSS

CSS (сокращение от *Cascading Style Sheets* — каскадные таблицы стилей) — фундаментальная технология, используемая для оформления, форматирования, позиционирования элементов, а также адаптации веб-страниц под различные размеры экранов. Она позволяет управлять визуальным оформлением HTML-элементов, таких как шрифты, цвета, макет и позиционирование.

Далее, рассмотрим основные правила CSS, позиционирование элементов и концепцию адаптивности веб-страницы.

CSS-стили могут быть подключены на страницу тремя разными вариантами (рис. 1):

- внутри тега <style> в секции <head> документа HTML (внутренние стили);
- внутри атрибута “style” любого тега HTML (инлайн-стили);
- подключение внешнего файла со стилями (внешние стили).

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="css/reset.css" />
    <link rel="stylesheet" href="css/base.css" />
    <link rel="stylesheet" href="css/main.css" />
    <link href="https://unpkg.com/boxicons@2.1.4/css/boxicons.min.css" rel="stylesheet">
    <title>Certification</title>
</head>
```

Рис. 1. Подключение внешних стилей

Каждый из этих способов имеет свои преимущества и недостатки. Но на практике удобнее использовать внешние стили (подключение внешних css-файлов), поскольку в этом случае мы не загромождаем разметку документа лишней информацией, а также имеем возможность быстро изменять стилизацию любых элементов на странице.

5.1. Правила CSS

CSS использует набор правил для определения того, как элементы должны быть оформлены. Каждое правило состоит из селектора и блока, в котором содержится описание стилей для данного селектора. Селектор нацелен на один или несколько HTML-элементов, а блок объявления содержит одну или несколько пар «свойство — значение».

Пример: допустим, у нас есть HTML-элемент с классом “box”. Мы можем стилизовать его с помощью CSS следующим образом:

```
***
.box {
    background-color: blue;
    color: white;
    font-size: 18px;
}
                                                    ***
```

В этом примере используется селектор по классу “.box”, который содержит три пары «свойство — значение». Свойство “background — color” устанавливает синий цвет фона, свойство “color” устанавливает белый цвет текста, а свойство “font-size” устанавливает размер шрифта, равным 18 пикселям.

5.2. Обнуляющий файл стилей

Каждый браузер имеет свои собственные параметры стилей по умолчанию. Если добавить на страницу код (представлен ниже) и запустить его, то в инструментах разработчика мы

сможем увидеть, что некоторые css-стили уже присутствуют на странице (рис. 2).

```
***
<h1>Heading1</h1>
<p>Paragraph</p>
<ul>
  <li>List1</li>
  <li>List2</li>
  <li>List3</li>
</ul>
<h2>Heading2</h2>
```

***

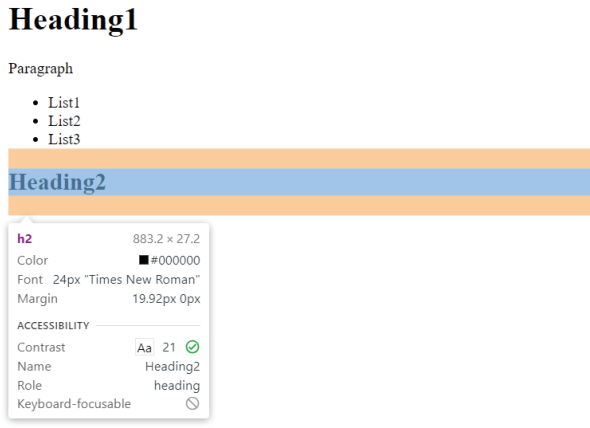


Рис. 2. Стили браузера *Chrome* по умолчанию

Чтобы отменить стили по умолчанию, обычно используют дополнительный файл, который называется “reset.css”. Он может выглядеть следующим образом:

```
***
/* http://meyerweb.com/eric/tools/css/reset/
    v2.0 | 20110126
    License: none (public domain)
*/
```

```
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,
figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary,
time, mark, audio, video {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    font: inherit;
    vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
    display: block;
}
body {
    line-height: 1;
}
ol, ul {
    list-style: none;
}
blockquote, q {
    quotes: none;
}
blockquote: before, blockquote: after,
q: before, q: after {
    content: '';
    content: none;
```

```
}
table {
    border-collapse: collapse;
    border-spacing: 0;
}
```

* * *

После подключения к веб-странице файла “reset.css” настройки стилей по умолчанию в браузере будут отменены, и появится возможность использовать свои собственные css-стили без какого-либо влияния на них со стороны встроенных параметров браузера.

5.3. Часто используемые свойства CSS

Рассмотрим часто используемые свойства CSS.

Font-family. Устанавливает семейство шрифта, которое будет использоваться для оформления элемента. По умолчанию используется семейство *Times New Roman*. Его мы и увидим при добавлении любого текста на страницу.

Семейство шрифта наследуется элементами на странице, поэтому очень часто это свойство пишут для тега <body> в самом начале файла со стилями (рис. 3). На странице может быть использовано несколько шрифтов одновременно.

```
15  body {
16    font-size: calc(20px + 12 * (100vw / 1920));
17    font-weight: 500;
18    color: #121212;
19    font-family: Roboto, sans-serif;
20    box-sizing: border-box;
21  }
```

Рис. 3. Установка *font-family* для веб-страницы

В настоящее время для подключения необходимых шрифтов чаще всего используют сервис *Google Fonts*[1].

[1] Google Fonts. URL: https://fonts.google.com/ (дата обращения: 02.02.2024).

Font-size. Определяет размер шрифта элемента. Добавим на страницу параграф, пропишем для него класс и укажем размер шрифта. Для этого достаточно набрать символы “fz30”, и мы получим необходимое нам свойство и его значение:

```
***
<p class=«text30»>Lorem ipsum dolor sit amet
consectetur adipisicing elit. Dolorum sapiente,
consequuntur, eius inventore praesentium est
nulla perspiciatis nemo eum cupiditate, nam
asperiores doloremque repudiandae? Illum hic id
minima quibusdam ut.</p>

.text30{
  font-size: 30px;
}
                                                ***
```

Размер шрифта можно устанавливать не только в пикселях, но и в относительных единицах, например, “em” (единица, основанная на размере шрифта родительского элемента) или “rem” (единица, основанная на размере шрифта корневого элемента HTML-документа). Например, если указать размер шрифта для “body” в “14px”, то значение “1.5rem” для какого-либо конкретного элемента сделает шрифт в полтора раза больше. По умолчанию, если размер шрифта не прописан, в браузерах используется “font-size: 14px”.

Font-style. Определяет начертание шрифта — обычное, курсивное или жирное. Также можно задать значение свойства “normal”, что вернет текст в состояние по умолчанию.

Font-weight. Задает насыщенность шрифта или его «вес». Значения могут быть как буквенные, так и указываться цифрами. Значение 400 является значением по умолчанию, его писать необязательно. Если необходимо использовать жирный шрифт, то можно использовать либо значение “bold”, либо соответствующее ему значение в цифрах — 700.

Color. Задает цвет шрифта внутри элементов. Можно использовать различные варианты: обычное текстовое значение (*white,*

black, green), шестнадцатеричное число (например, #a23f3f), формат “rgb” (например, rgb(74, 146, 74)) или в формате “rgba” (например, rgba(74, 146, 74, 0.5)). Последний вариант дает возможность также указать прозрачность применяемого цвета (альфа канал, *opacity*).

Также существует еще одна форма задания цвета (рис. 4): HSL (*hue*, *saturation*, *lightness*).

Hue — цветовой тон. Это цвет, с которым вы будете работать. Задается как градус на цветовом круге от 0 до 360. 0 — красный, 120 — зеленый и 240 — синий.

Saturation — насыщенность. Показывает то, насколько цвет сочный или приглушенный. Задается в процентах. 0 % означает оттенок серого, а 100 % — это полный цвет.

Lightness — яркость или светимость. Определяет, насколько цвет будет светлый и нежный или темный и глубокий. Задается в процентах. 0 % — черный, 50 % — ни светлый, ни темный, 100 % — белый.

Рис. 4. Установка цвета в формате HSL

Text-align. Определяет горизонтальное выравнивание текста. Возможные значения: *center*, *justify*, *right*, *left*.

Text-decoration. Добавляет различные виды подчеркивания к тексту. Текст можно сделать подчеркнутым, сделать линию сверху, а также перечеркнуть его. Одновременно можно применить сразу несколько значений, записав их через запятую.

Text-shadow. Добавляет к тексту тень. Синтаксис: смещение по горизонтали, смещение по вертикали, размер размытия и цвет тени (рис. 5). Запись будет выглядеть следующим образом:

```
***
h1 {
  text-shadow: 2px 2px 5px red;
}
                                    ***
```

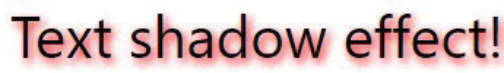


Рис. 5. Установка тени для текста

Text-transform. Преобразует текст в заглавные или прописные буквы (рис. 6).

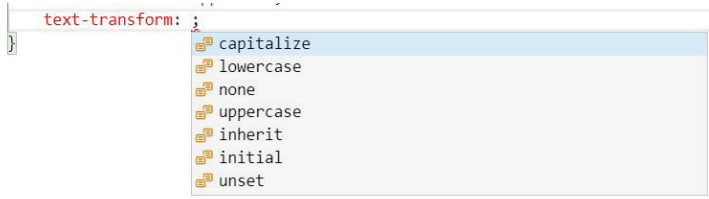


Рис. 6. Возможные значения свойства “text-transform”

Text-indent. Создает отступ первой строки (красная строка).

Letter-spacing и **word-spacing**. Определяет интервал между буквами и добавляет разреженность.

White-space. Управляет пробелами между словами. Обычно применяется со значением “nowrap”, которое запрещает перенос строки, и весь текст остается в одной строке. Значение “normal” устанавливает значение по умолчанию.

Line-height. Задает высоту строки. По умолчанию она равна единице. В верстке веб-страниц применяется очень часто.

Padding. Внутренний отступ блочных тегов:

```
***
.block{
  border: solid 2px green;
  padding: 10px 10px 10px 10px;
}
                                    ***
```

В примере выше отступы прописываются по часовой стрелке для каждой из четырех сторон блока: сверху, справа, снизу, слева.

Возможны и другие варианты записи. Значения: сверху, слева и справа, снизу:

```
***
.block{
  border: solid 2px green;
  padding: 10px 5px 10px;
}
                                                    ***
```

Значения: сверху и снизу, слева и справа:

```
***
.block{
  border: solid 2px green;
  padding: 10px 10px;
}
                                                    ***
```

Если указать одно значение, то отступы со всех четырех сторон будут одинаковыми. Значения можно задавать в процентах.

Margin. Внешние отступы элементов. Запись свойства аналогична “padding”. Здесь можно давать отрицательное значение и значения в процентах.

Width и Height. Высота и ширина элементов. Можно задавать в пикселях, процентах, а также относительных единицах:

- vh (1vh — 1 % от высоты *viewport*);
- vw (1vw — 1 % от ширины *viewport*).

Это свойство неприменимо к строчным тегам (кроме “img”). Чаще всего высоту элементов не указывают вообще, просто добавляя внешние и внутренние отступы сверху и снизу.

Max-width, min-width. Это некоторый ограничитель ширины блока. Если установить какое-либо значение, то оно будет действовать тогда, когда размер окна браузера больше или меньше этого значения:

```
***
.block{
  max-width: 500px;
}
                                                    ***
```

Border-box. Задает тип расчета ширины и высоты блоков без учета толщины границ. Если его не применять, то отступы будут включаться в расчет, и верстка будет вести себя непредсказуемо.

Overflow. Дает возможность показывать или не показывать содержимое блока, если оно больше размера блока. Значения могут быть следующими: *overflow*, *scroll*, *hidden*, *auto* (подключает полосы прокрутки только тогда, когда они нужны).

Display. Указывает, в каком качестве будет использоваться элемент. Возможные значения:

- *block* — элемент становится блочным, и к нему в этом случае можно применять все свойства блочных тегов;
- *inline* — элемент становится строчным, на него перестают действовать свойства блочных элементов;
- *inline-block* — элемент становится строчным, но все также занимает ширину, равную контенту. При этом на элемент действуют свойства высоты, ширины и отступов;
- *none* — удаляет элемент из верстки; часто с этим свойством взаимодействуют через *JavaScript*;
- *flex* — дает возможность применить к элементу свойства технологии *Flex-Box* для создания сложных гибких макетов (более подробно можно посмотреть по ссылке: https://www.w3schools.com/cssref/css3_pr_flex.php).

Border. Устанавливает границу блока. Задается следующим образом:

```
***
border: 1px solid black;
                                                    ***
```

В примере выше для свойства “border” указываются толщина, тип начертания и цвет границы.

Border-radius. Также мы можем задать радиус скругления границ блока. Для этого необходимо задать свойство “border-radius”, например:

`border-radius: 50px 0 50px 0;` (задается радиус для каждой из четырех сторон)

`border-radius: 50%;` (задается радиус в процентах для всех четырех сторон)

Outline. Устанавливает внешнюю границу блока. В отличие от свойства “border” устанавливается за границей блока.

Box-shadow. Устанавливает тень для блочного элемента:

```
***
.string{
  display: inline-block;
  padding: 20px;
  border: 1px solid green;
  box-shadow: 0px 0px 10px 5px black inset;
}
                                                    ***
```

Opacity. Задает прозрачность элемента. Не удаляет блок из верстки, а просто делает его прозрачным. Можно записать следующим образом:

```
***
.string{
  display: inline-block;
  padding: 20px;
  border: 1px solid green;
  box-shadow: 0px 0px 10px 5px black inset;
  opacity: 0.9;
}
                                                    ***
```

Это свойство часто анимируется для создания эффекта перехода.

Visibility. Устанавливает видимость элемента. В отличии от “display: none” блок остается в верстке, но с ним нельзя взаимодействовать. Часто используется для анимации элементов в скриптах.

Background. Управляет фоном элемента. Позволяет задать сразу несколько параметров для фона элемента: определить его цвет, установить фоновое изображение, задать начальную по-

зицию фонового изображения, определить, будет ли оно повторяться, будет ли прокручиваться фоновое изображение, масштабировать фоновое изображение согласно размерам.

Для задания фона можно, например, вырезать часть какой-либо картинки по вертикали и заполнить ей весь фон.

Также значением этого свойства может быть указан градиент. Градиент можно создать, используя сервис в сети Интернет, и вставить в файл стилей готовый код (https://www.colorzilla.com/gradient-editor/).

Синтаксис свойства следующий:

```
***
background: url(« «) 0 0 no-repeat fixed;
                                                    ***
```

Есть возможность выставить положение фона в пикселях или значениями “Top”, “Left”, “Right”, “Bottom”:

```
***
.block{
  width: 500px;
  height: 300px;
  padding: 30px;
  padding-left: 120px;
  background: url("../img/bg.png") 20px 20px no-repeat;
}
                                                    ***
```

Оформление текста. CSS позволяет применять к тексту различные стили, такие как изменение шрифта, цвета, размера и выравнивания.

Пример:

```
***
h1 {
  font-family: Arial, sans-serif;
  color: #FF0000;
  font-size: 24px;
```

```
  text-align: center;
}
```

***

В этом примере элемент заголовка “h1” будет иметь семейство шрифтов *Arial*, красный цвет (#FF0000), размер шрифта 24 пикселя и будет расположен по центру.

Задание теней элементам. Тени в CSS добавляют элементам глубину. Вы можете управлять цветом, размером и положением тени.

Пример:

```
***
.box {
  box-shadow: 2px 2px 5px rgba(0, 0, 0, 0.3);
}
```

***

В этом примере элемент “.box” будет иметь тень со смещением по горизонтали на 2 пикселя, смещением по вертикали на 2 пикселя, радиусом размытия в 5 пикселей и слегка прозрачным черным цветом (rgba(0, 0, 0, 0.3)).

Эффекты наведения (*Hover Effects*). CSS позволяет добавлять эффекты к элементам, которые меняют свой внешний вид при наведении на них курсора. Это достигается путем использования свойства “transition”, которое дает возможность указать, какие именно свойства элемента необходимо изменять при наведении на элемент курсора мыши. Вы можете указать длительность изменения значения, временную функцию, задержку.

Синтаксис свойства “transition” показан ниже:

```
***
transition: [none | <transition-property>] ||
<transition-duration> || <transition-timing-
function> || <transition-delay>;
```

***

Пример:

```
***
.button {
```

```
  background-color: #007BFF;
  color: #FFFFFF;
  padding: 10px 20px;
  transition: background-color 0.3s ease;
}

.button: hover {
  background-color: #0056b3;
}
```

***

В этом примере элемент ".button" будет иметь синий цвет фона и белый текст. При наведении курсора цвет фона плавно перейдет в более темный оттенок синего.

5.4. Позиционирование элементов

CSS предоставляет несколько способов позиционирования элементов на веб-странице. Наиболее часто используемыми методами позиционирования являются "static", "relative", "absolute" and "fixed":

- *static positioning* — это расположение элементов HTML по умолчанию, когда элементы выводятся на страницу в том порядке, в котором они идут в исходном HTML-коде. Такой порядок еще называют «нормальный поток»;
- *relative positioning* — элементы, расположенные относительно друг друга, регулируются относительно их нормального положения. Вы можете использовать такие свойства, как *top*, *bottom*, *left* и *right*, чтобы сместить элемент от его нормального положения;
- *absolute positioning* — при абсолютном позиционировании элементы располагаются относительно их ближайшего родительского элемента. Если ни у одного из родительских элементов нет определенной позиции, она будет находиться относительно самого документа;

– *fixed positioning* — фиксирует положение элемента относительно окна браузера. Он остается в том же положении даже при прокрутке страницы.

Пример. Рассмотрим следующий фрагмент кода CSS:

```
***
.box {
  position: relative;
  top: 20px;
  left: 30px;
}
                                                    ***
```

В этом примере элемент ".box" расположен относительно, и он будет перемещен на 20 пикселей вниз и на 30 пикселей вправо от своего обычного положения.

5.5. Flexbox Layout

CSS Flexbox предоставляет гибкий способ компоновки элементов внутри контейнера, обеспечивая легкое выравнивание и распределение элементов на странице.

Пример:

```
***
.container {
  display: flex;
  justify-content: space-between;
}

.item {
  flex: 1;
  margin: 10px;
}
                                                    ***
```

В этом примере элемент ".container" расположит свои дочерние элементы с равным расстоянием между ними по гори-

зонтали. Элементы ".item" будут иметь одинаковое значение свойства "flex grow", в результате чего они будут занимать равное пространство внутри контейнера.

Эти примеры демонстрируют лишь некоторые из множества возможностей, которые предлагает CSS для стилизации и верстки веб-страниц. Эксперименты с различными свойствами и значениями помогут вам получить более глубокое представление о CSS и его возможностях.

5.6. Responsive Web Design

Responsive Web Design (отзывчивый веб-дизайн, также известный как *responsive layout*) — это практика проектирования веб-страниц, которые адаптируются к различным размерам экрана и устройствам и реагируют на них. Это гарантирует, что содержимое и макет остаются читаемыми и пригодными для использования на различных устройствах, таких как настольные компьютеры, ноутбуки, планшеты и смартфоны.

Для создания отзывчивого макета необходимо учитывать несколько ключевых принципов и приемов.

Fluid Grids. Одной из фундаментальных концепций является использование *Fluid Grids*. Вместо использования фиксированных измерений ширины элементов на основе пикселей в *Fluid Grids* используются относительные единицы измерения, такие как проценты. Это позволяет пропорционально изменять размер элементов в зависимости от доступной ширины экрана.

Пример:

```
***
.container {
  width: 100%;
  display: grid;
  grid-template-columns: repeat(auto-fit,
minmax(200px, 1fr));
  gap: 20px;
}
                                                ***
```

В этом примере элемент ".container" использует макет сетки со столбцами, которые динамически настраиваются в зависимости от доступной ширины. Значение "repeat(autofit, min max(200px, 1fr))" гарантирует, что каждый столбец будет иметь минимальную ширину в 200 пикселей, но будет расширяться, чтобы равномерно заполнить любое дополнительное пространство.

Media Queries (*медиазапросы*). Медиазапросы — это мощная функция CSS, которая позволяет применять различные стили в зависимости от конкретных условий, таких как ширина и высота экрана, ориентация или тип устройства. Медиазапросы позволяют создавать точки изменения значений "breakpoints", в которых макет и стили могут быть соответствующим образом скорректированы.

Пример:

```
***
@media screen and (max-width: 768px) {
  .container {
    grid-template-columns: repeat(auto-fit,
minmax(150px, 1fr));
  }
}
                                                ***
```

В этом примере медиазапрос нацелен на экраны с максимальной шириной 768 пикселей. Внутри запроса столбцы элемента ".container" настраиваются таким образом, чтобы их минимальная ширина составляла 150 пикселей, обеспечивая более подходящий макет для экранов меньшего размера.

Flexible Images и *Media*. Отзывчивые макеты также должны учитывать свойства изображений и мультимедийных элементов. Чтобы они не увеличивались или не становились слишком маленькими, можно применить такие свойства CSS, как "max-width: 100 %" и "height: auto", чтобы гарантировать, что изображения и мультимедийные материалы надлежащим образом масштабируются в своих контейнерах.

Пример:

```
***
img {
  max-width: 100%;
  height: auto;
}
                                                    ***
```

Код CSS выше гарантирует, что изображения никогда не будут превышать ширину своего контейнера, сохраняя соотношение сторон и уменьшая масштаб при необходимости.

Mobile-First Approach. Одной из распространенных стратегий в разработке отзывчивых макетов является использование подхода, ориентированного в первую очередь на мобильные устройства. Данный подход предусматривает первоначальное проектирование и стилизацию макета для экранов меньшего размера, а затем постепенную адаптацию макета для экранов бóльшего размера. Начав с минималистичного и удобного для мобильных устройств макета, вы гарантируете, что контент останется доступным на всех устройствах.

Отзывчивый дизайн необходим для создания удобных и доступных веб-страниц на разных устройствах. Используя гибкие сетки и изображения, медиазапросы и применяя подход, ориентированный на мобильные устройства, вы можете создавать макеты, которые легко адаптируются и обеспечивают оптимальный пользовательский опыт.

Ниже представлена обновленная версия кода (“Pizza Order”) с добавленным разделом галереи и примененными дополнительными стилями CSS:

```
***
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">
```

```
<title>Pizza Order</title>
<style>
  /* Basic styling */
  body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 20px;
  }

  /* Header styles */
  header {
    background-color: #f2f2f2;
    padding: 10px;
    text-align: center;
  }

  nav ul {
    list-style: none;
    padding: 0;
    margin: 0;
    display: flex;
    justify-content: center;
  }

  nav ul li {
    margin: 0 10px;
  }

  nav ul li a {
    text-decoration: none;
    color: #333;
  }

  /* Main styles */
  main {
    margin-bottom: 20px;
  }
```

```
section {
  padding: 10px;
  background-color: #f9f9f9;
  margin-bottom: 10px;
}

section h2 {
  color: #333;
}

section p {
  color: #666;
}

ul {
  list-style: disc;
  margin-left: 20px;
}

/* Form styles */
form label {
  display: block;
  margin-bottom: 5px;
  color: #333;
}

form input[type="text"],
form textarea {
  width: 100%;
  padding: 5px;
  margin-bottom: 10px;
  border: 1px solid #ccc;
}

form input[type="submit"] {
  background-color: #333;
  color: #fff;
  padding: 10px 20px;
```

```
      border: none;
      cursor: pointer;
    }

    /* Footer styles */
    footer {
      background-color: #f2f2f2;
      padding: 10px;
      text-align: center;
    }

    /* Gallery styles */
    .gallery {
      display: flex;
      flex-wrap: wrap;
      justify-content: center;
    }

    .gallery-item {
      width: 200px;
      margin: 10px;
    }

    .gallery-item img {
      width: 100%;
      height: auto;
    }

    /* Media queries */
    @media (max-width: 767px) {
      .gallery-item {
        width: 100%;
        max-width: 300px;
      }
    }
  </style>
</head>
<body>
```

```
  <header>
    <h1>Pizza Order</h1>
    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">Menu</a></li>
        <li><a href="#">Order</a></li>
        <li><a href="#">Contact</a></li>
      </ul>
    </nav>
  </header>

  <main>
    <section>
      <h2>Choose Your Pizza</h2>
      <p>Select your favorite pizza from our menu:</p>
      <div class="gallery">
        <div class="gallery-item">
          <img src="pizza1.jpg" alt="Margherita Pizza">
          <p>Margherita Pizza</p>
        </div>
        <div class="gallery-item">
          <img src="pizza2.jpg" alt="Pepperoni Pizza">
          <p>Pepperoni Pizza</p>
        </div>
        <div class="gallery-item">
          <img src="pizza3.jpg" alt="Hawaiian Pizza">
          <p>Hawaiian Pizza</p>
        </div>
        <div class="gallery-item">
          <img src="pizza4.jpg" alt="Veggie
Supreme Pizza">
          <p>Veggie Supreme Pizza</p>
        </div>
      </div>
    </section>

    <section>
```

```
      <h2>Customize Your Pizza</h2>
      <p>Customize your pizza with the following
options:</p>
      <ul>
        <li>Size:
          <select>
            <option>Small</option>
            <option>Medium</option>
            <option>Large</option>
          </select>
        </li>
        <li>Toppings:
          <label><input type="checkbox">
Pepperoni</label>
          <label><input type="checkbox">
Mushrooms</label>
          <label><input type="checkbox"> Olives</
label>
        </li>
        <li>Crust:
          <label><input type="radio" name="crust"
value="thin"> Thin</label>
          <label><input type="radio" name="crust"
value="thick"> Thick</label>
        </li>
      </ul>
    </section>

    <section>
      <h2>Place Your Order</h2>
      <p>Enter your details and submit your order:</p>
      <form>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name"
required>
        <br>
        <label for="address">Address:</label>
        <textarea id="address" name="address"
```

```
required></textarea>
        <br>
        <input type="submit" value="Submit Order">
      </form>
    </section>
  </main>

  <footer>
    &copy; 2023 Pizza Order. All rights reserved.
  </footer>
</body>
</html>
```

***

В коде выше мы добавили раздел галереи, используя элемент , и применили стили CSS для создания отзывчивого макета галереи. Каждый элемент пиццы представлен контейнером с классом “gallery-item”, содержащим тег <img> для изображения пиццы и тег <p> для названия пиццы. Вы можете заменить URL-адреса изображений (*pizza1.jpg*, *pizza2.jpg* и т. д.) на ваши собственные URL-адреса изображений или используйте соответствующие пути к файлам изображений.

Также были внесены следующие изменения:

- добавлены стили “Flexbox” в класс “.gallery” для создания гибкого макета галереи с переносом элементов;
- увеличена ширина элемента “.gallery” до 200 пикселей и добавлено поле для интервала;
- добавлен медиазапрос с использованием “@media” для изменения ширины элемента “.gallery” до 100 % с максимальной шириной 300 пикселей, когда ширина области просмотра составляет 767 пикселей или меньше. Это гарантирует, что элементы галереи будут располагаться вертикально на экранах меньшего размера.

Вопросы для самопроверки

1. Как расшифровывается аббревиатура CSS?
2. Какие основные задачи решает CSS в веб-разработке?
3. Каким образом можно добавить CSS-стили к HTML-документу?
4. Как создать правило CSS для изменения цвета текста?
5. Как изменить размер и тип шрифта с помощью CSS?
6. Как задать цвет фона для элемента с помощью CSS?
7. Какие единицы измерения используются для задания размеров в CSS?
8. Какие свойства и значения CSS используются для управления отступами и положением элементов на странице?
9. Как создать класс CSS и как применить его к элементам на странице?
10. Что такое «классы» и «идентификаторы» (id) в CSS, и в чем различие между ними?
11. Какие свойства CSS используются для создания анимаций и переходов (*transitions*)?
12. Как реализовать адаптивный дизайн с помощью медиазапросов (*media queries*)?

6. Bootstrap Framework

Bootstrap — это популярный интерфейсный фреймворк для создания отзывчивых (адаптивных) веб-страниц. Ниже мы рассмотрим основы *Bootstrap* и несколько примеров, которые помогут вам начать создавать макеты с использованием этого фреймворка.

6.1. Что такое Bootstrap?

Bootstrap — это CSS-фреймворк с открытым исходным кодом, разработанный *Twitter*. Он предоставляет набор готовых отзывчивых компонентов, таких как сетки, типографика, формы, кнопки и навигация. *Bootstrap* позволяет разработчикам быстро создавать современные и визуально привлекательные веб-сайты и приложения.

Основные особенности *Bootstrap*:

- *адаптивная система сеток* — система сеток *Bootstrap* основана на макете из 12 столбцов, что позволяет разработчикам создавать адаптивные дизайны для экранов различных размеров;
- *компоненты CSS* — включает в себя широкий спектр повторно используемых компонентов CSS, таких как кнопки, формы, панели навигации и оповещения;
- *типографика* — предоставляет набор типографских стилей и утилит для быстрого форматирования текста;
- *плагины JavaScript* — поставляется в комплекте с плагинами *JavaScript*, такими как карусели, модальные окна,

всплывающие подсказки и многое другое, для улучшения функциональности веб-страницы;
– *настройка — Bootstrap* можно настроить в соответствии с конкретными требованиями к дизайну, изменив переменные или используя пользовательские таблицы стилей.

6.2. Начало работы с Bootstrap

1. Загрузка *Bootstrap*. Посетите официальный веб-сайт *Bootstrap*[1] и загрузите предварительно скомпилированные файлы CSS и *JavaScript*.

2. Подключение файлов *Bootstrap*. Добавьте ссылки на файлы *Bootstrap* CSS в раздел “head” и ссылки на файлы *JavaScript* непосредственно перед закрывающим тегом “body” в HTML-файле.

3. Использование *Bootstrap CDN*. В качестве альтернативы вы можете использовать сеть доставки контента *Bootstrap* (CDN), добавив соответствующие ссылки на файлы CSS и *JavaScript* в раздел “head” HTML-файла.

6.3. Grid System (система сеток)

Система сеток *Bootstrap* обеспечивает гибкий и отзывчивый макет для создания веб-страниц. Основана на макете из 12 столбцов, который позволяет разделить страницу на несколько столбцов.

Примеры:

: создает контейнер, в котором размещается система сеток;

: определяет строку для хранения столбцов;

: создает столбец шириной 6 из 12 столбцов на экранах среднего размера.

[1] Bootstrap : [website]. URL: getbootstrap.com (date of accessed: 02.02.2024).

6.4. CSS компоненты

Bootstrap предлагает широкий спектр CSS-компонентов, которые могут быть легко интегрированы в веб-страницы.

Примеры:

Buttons: <button class="btn btn-primary">Primary Button</button>

Forms: <input class="form-control" type="text" placeholder="Enter your name">

Navigation: <nav class="navbar navbar-expand-lg navbar-dark bg-dark">...</nav>

Alerts: ...

6.5. Типографика

Bootstrap предоставляет набор предопределенных стилей типографики и утилит.

Примеры:

Заголовки: <h1>Heading 1</h1> to <h6>Heading 6</h6>

Форматирование текста: This is a lead paragraph.

Выравнивание текста: Center-aligned text

6.6. JavaScript-плагины

Bootstrap включает в себя множество плагинов *JavaScript*, которые могут расширить функциональность веб-страницы.

Примеры:

Карусели (*слайдеры*): ...

Модальные окна: ...

Подсказки: <button class="btn" data-toggle="tooltip" data-placement="top" title="Tooltip text">...</button>

6.7. Кастомизация Bootstrap

Bootstrap может быть настроен в соответствии с вашими требованиями к дизайну.

Параметры настройки включают в себя:

- *изменение переменных* — *Bootstrap* предоставляет набор SCSS-переменных, которые можно переопределять для изменения цветов, интервалов и многого другого;
- *пользовательские таблицы стилей* — вы можете создавать пользовательские таблицы стилей CSS и переопределять стили *Bootstrap* по умолчанию.

Таким образом, *Bootstrap* — это мощный фреймворк, который позволяет быстро создавать адаптивные веб-страницы. Он предоставляет широкий спектр готовых компонентов и адаптивную систему сеток. Используя *Bootstrap*, вы можете сэкономить время и усилия при проектировании и разработке современных веб-сайтов и приложений.

Более подробную информацию по использованию *Bootstrap Framework* можно найти по следующим ссылкам:

- официальный веб-сайт: https://getbootstrap.com/;
- документация: https://getbootstrap.com/docs/;
- примеры: https://getbootstrap.com/docs/examples/.

Вопросы для самопроверки

1. Что такое *Bootstrap*, и какую роль он играет в веб-разработке?
2. Как подключить *Bootstrap* к вашему проекту?
3. Какие классы используются для создания столбцов и рядов в *Bootstrap Grid System*?

4. Как создать кнопку с использованием *Bootstrap*, и какие стили она будет иметь по умолчанию?
5. Какие классы *Bootstrap* используются для создания навигационного меню?
6. Какие компоненты *Bootstrap* предоставляются для работы с иконками и шрифтами?
7. Какие JavaScript-компоненты предоставляются *Bootstrap*?
8. Какие встроенные CSS-классы *Bootstrap* можно использовать для стилизации текста?

7. Язык программирования JavaScript

JavaScript — это кроссплатформенный объектно-ориентированный язык программирования, используемый разработчиками для придания интерактивности веб-страницам. Он позволяет разработчикам создавать динамически обновляемый контент, использовать анимацию, всплывающее меню, интерактивные кнопки, управлять мультимедиа и т. д.

Далее, рассмотрим фундаментальные концепции *JavaScript* и примеры, которые помогут понять, каким образом он используется.

7.1. Переменные и типы данных

JavaScript позволяет хранить данные и манипулировать ими с помощью переменных.

Пример:

```
***
// Declaration and assignment of variables
let name = "John";
let age = 25;
let isStudent = true;
                                                    ***
```

В этом примере мы объявляем три переменные: “name”, “age” и “isStudent”. Мы присваиваем этим переменным значения “John” (строка), “25” (число) и “true” (логическое значе-

ние) соответственно. *JavaScript* поддерживает различные типы данных, включая строки, числа, логические значения, массивы, объекты и многое другое.

7.2. Операторы и выражения

JavaScript предоставляет ряд операторов для выполнения арифметических, сравнительных и логических операций.

Давайте рассмотрим несколько примеров:

```
***
let a = 5;
let b = 10;
let sum = a + b; // Addition
let product = a * b; // Multiplication
let isGreater = a > b; // Comparison
let logicalAnd = (a > 0) && (b > 0); // Logical
AND
                                                ***
```

В этом примере мы выполняем сложение, умножение, сравнение и логические операции AND, используя переменные “a” и “b”.

7.3. Управление потоком (ветвление)

JavaScript позволяет вам управлять ветвлением программы с помощью условных операторов и циклов. Рассмотрим несколько примеров:

Условные операторы (“if – else”):

```
***
let age = 18;
if (age >= 18) {
  console.log("You are eligible to vote.");
} else {
```

```
  console.log("You are not eligible to vote yet.");
}
```

***

В этом примере мы проверяем, превышает ли возраст 18 лет или равен ему. Если это так, мы печатаем сообщение о праве на участие в голосовании; в противном случае мы печатаем другое сообщение.

Циклы (“for loop”):

```
***
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

***

Этот цикл выводит числа от 0 до 4 на экран. Цикл начинается с i = 0, проверяет условие (i < 5), выполняет код внутри цикла и увеличивает “i” на 1 до тех пор, пока условие не перестанет быть истинным.

7.4. Функции

Это повторно используемые блоки кода в *JavaScript*. Они позволяют определить набор инструкций, которые могут быть вызваны несколько раз. Например:

```
***
function greet(name) {
  console.log("Hello, " + name + "!");
}

greet("John");
```

***

В этом примере мы определяем функцию с названием “greet”, которая принимает параметр “name” и печатает приветственное сообщение. Затем мы вызываем функцию с аргументом “John”, в результате чего получаем вывод “Hello, John!”

Таким образом, *JavaScript* — это универсальный язык программирования, который является неотъемлемой частью веб-разработки. Выше мы рассмотрели некоторые основные понятия, включая переменные, типы данных, операторы, поток управления и функции.

7.5. Ресурсы для изучения JavaScript

Ниже представлены ресурсы, которые могут помочь вам изучить *JavaScript* на более профессиональном уровне.

1. *Mozilla Developer Network (MDN) JavaScript Guide*. Веб-сайт: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide. Руководство MDN по *JavaScript* — отличный ресурс для начинающих. В нем рассматриваются основы *JavaScript*, а также содержатся подробные объяснения и примеры.
2. *JavaScript.info*. Веб-сайт: https://JavaScript.info/. *JavaScript.info* — это ресурс для изучения *JavaScript* с нуля. Здесь рассматриваются темы, начиная от основ и заканчивая более продвинутыми концепциями, включая объектно-ориентированное программирование и асинхронное программирование. Веб-сайт предлагает интерактивные примеры и упражнения для углубленного понимания языка.
3. Книга "*Eloquent JavaScript*" автора *Marijn Haverbeke*. Эта книга высоко ценится в сообществе *JavaScript*. Она предлагает для начинающих удобное введение в *JavaScript*, охватывающее фундаментальные концепции и постепенно переходящее к более продвинутым темам. Книга включает в себя практические упражнения для закрепления полученных знаний.
4. Книга "*JavaScript: The Good Parts*" автора *Douglas Crockford*. Эта книга посвящена основным разделам *JavaScript*, содержит рекомендации по наилучшим практикам и позволяет избежать распространенных ошибок при написании кода. Это краткий и содержательный ресурс для понимания основных концепций *JavaScript*.

5. *FreeCodeCamp*. Веб-сайт: https://www.freecodecamp.org/. *FreeCodeCamp* предлагает бесплатную интерактивную платформу для обучения веб-разработке, включая *JavaScript*. Он предоставляет структурированную учебную программу с практическими упражнениями по кодированию и проектами. Раздел *JavaScript* охватывает широкий спектр тем и подходит для начинающих.

6. *Codecademy*. Веб-сайт: https://www.codecademy.com/learn/introduction-to-JavaScript. *Codecademy* предлагает интерактивный курс *JavaScript*, который обучает основам языка. Курс предоставляет практический опыт обучения, позволяя писать код непосредственно в браузере и получать мгновенную обратную связь.

7.6. Примеры использования JavaScript

1. *Динамическое изменение содержимого HTML*. *JavaScript* можно использовать для манипулирования элементами HTML и динамического изменения их контента. Ниже показан пример, в котором мы изменяем текст элемента абзаца при нажатии кнопки:

```
***
<!DOCTYPE html>
<html>
<head>
  <title>JavaScript Example</title>
</head>
<body>
  <p id="demo">Click the button to change this
text.</p>
  <button onclick="changeText()">Click Me</
button>

  <script>
    function changeText() {
```

```
      document.getElementById("demo").innerHTML =
"Text changed!";
    }
  </script>
</body>
</html>
```

***

При нажатии на кнопку вызывается функция “changeText()”, которая выбирает элемент абзаца с идентификатором “demo” и изменяет его свойство “innerHTML” для отображения нового текста.

2. *Обработка отправки данных формы*. *JavaScript* можно использовать для проверки вводимых данных в форму и выполнения действий при ее отправке. Ниже показан пример, в котором проверяются данные, введенные пользователем в поля формы перед ее отправкой:

```
***
<!DOCTYPE html>
<html>
<head>
  <title>JavaScript Example</title>
</head>
<body>
  <form onsubmit="return validateForm()">
    <input type="text" id="name"
placeholder="Enter your name">
    <input type="email" id="email"
placeholder="Enter your email">
    <input type="submit" value="Submit">
  </form>

  <script>
    function validateForm() {
      var name = document.getElementById("name").
value;
      var email = document.
getElementById("email").value;
```

```
      if (name === "" || email === "") {
        alert("Please fill in all fields.");
        return false;
      }

      // Additional validation or form processing
code can be added here

      // The form will only be submitted if all
validations pass
      alert("Form submitted successfully!");
      return true;
    }
  </script>
</body>
</html>
```

***

Функция “validateForm()” вызывается при отправке формы. Она извлекает значения полей «имя» и «электронная почта», проверяет, пусты ли они. Если хотя бы одно из полей пустое, то выводится предупреждающее сообщение. Если все проверки пройдены, выводится предупреждение, указывающее на успешную отправку формы.

3. Создание простого слайдера. *JavaScript* можно использовать для создания интерактивных элементов, таких как слайдер. Ниже показан пример, в котором реализуется базовый слайдер с изображениями:

```
***
<!DOCTYPE html>
<html>
<head>
  <title>JavaScript Example</title>
  <style>
    .slideshow {
      max-width: 500px;
      height: 300px;
```

```
      position: relative;
      margin: 0 auto;
    }

    .slideshow img {
      width: 100%;
      height: 100%;
      object-fit: cover;
    }

  </style>
</head>
<body>
  <div class="slideshow">
    <img src="image1.jpg" id="slide"
alt="Slideshow Image">
  </div>

  <script>
    var images = ["image1.jpg", "image2.jpg",
"image3.jpg"];
    var currentIndex = 0;

    function changeSlide() {
      var slide = document.getElementById("slide");
      slide.src = images[currentIndex];

      currentIndex++;
      if (currentIndex >= images.length) {
        currentIndex = 0;
      }

      setTimeout(changeSlide, 2000); // Change
slide every 2 seconds
    }

    changeSlide(); // Start the slideshow
  </script>
```

```
</body>
</html>
```

* * *

В этом примере определен массив URL-адресов изображений и функция “changeSlide()”, которая используется для обновления атрибута “src” элемента “img”. Функция “setTimeout()” используется для повторного вызова “changeSlide()” каждые 2 секунды, создавая автоматический эффект слайд-шоу.

Это лишь несколько примеров того, как можно использовать *JavaScript* для добавления интерактивности на веб-страницы. Универсальность *JavaScript* позволяет использовать его для широкого спектра задач, включая проверку форм, анимацию, манипулирование DOM и многих других.

Вопросы для самопроверки

1. Что такое *JavaScript*, и какую роль он играет в веб-разработке?
2. Как вставить JavaScript-код в HTML-документ?
3. Какие базовые типы данных существуют в *JavaScript*, и какие операции можно выполнять с ними?
4. Какие ключевые слова используются для объявления переменных в *JavaScript*?
5. Какие операторы условия (“if”, “else”, “switch”) существуют в *JavaScript*, и как они работают?
6. Как создать циклы (“for”, “while”) в *JavaScript*, и как они используются для выполнения повторяющихся задач?
7. Что такое функции в *JavaScript*, и как их объявлять и вызывать?
8. Какие структуры данных (например, массивы, объекты, множества) используются в *JavaScript*, и как их использовать?
9. Какие популярные библиотеки и фреймворки используют *JavaScript* для разработки веб-приложений?

8. Библиотека React.js

Разработка любого веб-проекта включает в себя написание клиентской части приложения. Сегодня существует довольно много фреймворков и библиотек для решения этой задачи, например, *jQuery*, *React.js*, *Angular.js*, *Vue.js*. Некоторые из этих инструментов являются фреймворками, некоторые — библиотеками. Давайте посмотрим, в чем разница между ними.

Что такое фреймворк?

Любой разработчик рано или поздно сталкивается с двумя новыми для себя терминами: библиотека и фреймворк. Термины разные, но часто их значение считают одинаковым. Особенно, когда дело доходит до контекста интерфейса.

Слово «фреймворк» (*framework*) буквально переводится как «основа». Говоря еще проще: фреймворк — это фундамент, на котором может быть построено любое приложение. Для фреймворка не имеет значения, *что* вы будете разрабатывать с его помощью, важно *как*. Для этого в фреймворке есть специальные правила — ограничения.

Архитектура фреймворка может включать в себя файловую структуру проекта, встроенный набор инструментов для решения различных задач, набор внутренних абстракций, CLI-инструменты (*Command Line Interface*) для автоматизации задач и т. д.

Существует множество фреймворков для интерфейсной разработки. Из наиболее популярных выделяются *Angular*, *Vue* и *Ember*. Все они предоставляют набор встроенных инструмен-

тов, абстракций и т. д. Их можно использовать прямо из коробки, не тратя время на поиск подходящих модулей и налаживание связей.

Что такое библиотека?

Если *фреймворк* — это готовый инструмент для разработки приложения, то *библиотека* — это просто отдельный инструмент для решения какой-то задачи или нескольких задач. Библиотека не накладывает никаких ограничений на архитектуру или другие инструменты, используемые в проекте. Она сама по себе.

Не нужно далеко ходить за примерами. Некогда популярная библиотека *jQuery* — это не что иное, как набор функций, которые упрощают взаимодействие с DOM (объектная модель документа). Другой пример — *Lodash* — библиотека с обширным набором полезных функций на все случаи жизни.

Перечисленные выше библиотеки, как и любые другие, не предъявляют никаких требований к архитектуре или типу проекта. Неважно, как организован ваш проект, ничто не мешает вам включить библиотеку и использовать ее функциональность.

Библиотеку можно рассматривать как дополнительный модуль с полезными функциями. Опять же, библиотека — это инструмент. Вполне логично, что в одном проекте можно использовать несколько разных инструментов (библиотек), это нормальная ситуация. Однако при этом в проекте может быть только один фреймворк.

8.1. Что такое React.js?

React.js представляет собой библиотеку *JavaScript* с открытым исходным кодом, созданную *Facebook*[1]. Она обеспечивает декларативный и эффективный способы построения пользовательских интерфейсов путем разбиения их на повторно используемые компоненты.

[1] Деятельность компании *Meta Platforms, Inc.* запрещена на территории РФ.

8.2. Базовые концепции

Для того чтобы использовать *React.js*, важно понимать некоторые из ее основных концепций.

1. **Components**. *Компонент* — это повторно используемый автономный модуль, который отображает часть пользовательского интерфейса. Компоненты можно рассматривать как строительные блоки, которые можно комбинировать для создания более крупных и сложных пользовательских интерфейсов. Существует два типа компонентов: функциональные и классовые.
2. **JSX (JavaScript XML)**. *JSX* — это расширение синтаксиса, используемое *React*, которое позволяет писать подобный HTML код внутри кода *JavaScript*. Этот синтаксис позволяет описывать структуру компонента, как он должен выглядеть в окне браузера. JSX делает код более читабельным и интуитивно понятным, помогая преодолеть разрыв между HTML и *JavaScript*.
3. **Virtual DOM (Document Object Model)**. *React.js* использует виртуальный DOM для эффективного обновления и рендеринга компонентов. Виртуальный DOM — это облегченная копия реального DOM, которая представляет структуру веб-страницы. Когда состояние компонента изменяется, *React.js* сравнивает виртуальный DOM с фактическим DOM и обновляет только необходимые части, что обеспечивает оптимальную производительность.
4. **State**. *State* (состояние) представляет данные, которые компонент может хранить и которыми может управлять. Это позволяет компонентам вести себя динамично, обновляя и отражая изменения в пользовательском интерфейсе. Состояние обычно используется для хранения информации, которая может изменяться с течением времени, например, пользовательский ввод, ответы API или результат вычислений.

5. **Props (Properties)**. *Props* — это входные данные, которые компоненты получают от своих родительских компонентов. Они доступны только для чтения и позволяют передавать данные от родительского компонента к его дочерним элементам. *Props* необходимы для передачи информации и поведения между компонентами, что упрощает создание многоразового и модульного кода.

Как уже было сказано выше, *React.js* — это библиотека *JavaScript*, используемая для создания пользовательских интерфейсов. В отличие от фреймворков, *React.js* предоставляет основанный на компонентах подход к разработке пользовательского интерфейса. Его можно использовать для визуализации в сочетании с другими библиотеками. Например, *React Native* используется для разработки мобильных приложений, в то время как *React 360* используется для создания приложений виртуальной реальности.

Основная цель *React.js* — свести к минимуму ошибки при разработке пользовательского интерфейса. Это достигается за счет использования автономных и повторно используемых компонентов, которые описывают различные части пользовательского интерфейса. Эти компоненты могут быть объединены для формирования полноценного пользовательского интерфейса. Абстрагируясь от большей части процесса рендеринга, *React.js* позволяет разработчикам больше сосредоточиться на дизайне и функциональности своих приложений.

React.js запускается в браузере пользователя и манипулирует элементами страницы после их загрузки на стороне клиента. Это позволяет создавать довольно сложные интерактивные интерфейсы (например, как *Netflix.com*). Работа таких интерфейсов не предполагает постоянных запросов к серверу. Интерфейс работает эффективно и быстро, при этом DOM страницы обновляется в режиме реального времени, без обращения к серверу. Все это позволяет создать качественный пользовательский интерфейс, а также пользовательский опыт, и организовать работу с веб-страницей так, как если бы мы работали

с мобильным приложением, мгновенно переключаясь между страницами.

Мобильные приложения и SPA (одностраничные приложения) мгновенно реагируют на действия пользователя. Нет необходимости ждать загрузки страницы или начала выполнения какого-либо скрипта.

Кто-то может сказать: «Зачем нам нужен *React*, если мы можем просто использовать чистый *JavaScript*?» Ответ очень прост: *React.js* позволяет повторно использовать код для работы с определенным компонентом много раз. При использовании чистого JS вы должны описать каждый шаг, который необходимо выполнить для управления элементом пользовательского интерфейса (поиск элемента в DOM, создание кнопки, добавление класса, добавление *EventListener* и т. д.). *React.js* просто создает компонент и весь необходимый код для работы с этим компонентом один раз. Далее, вы просто добавляете этот компонент на страницу, и все работает.

Пример чистого JS:

```
***
const button = document.querySelector('button');

let modal;
let backdrop;

button.addEventListener('click', showModalHandler);

function showModalHandler() {
  if (modal) {
    return;
  }

  modal = document.createElement('div');
  modal.className = 'modal';

  const modalText = document.createElement('p');
  modalText.textContent = 'Are you sure?';
```

```
  const modalCancelAction = document.
createElement('button');
  modalCancelAction.textContent = 'Cancel';
  modalCancelAction.className = 'btn btn — alt';
  modalCancelAction.addEventListener('lick',
closeModalHandler);

  const modalConfirmAction = document.
createElement('button');
  modalConfirmAction.textContent = 'Confirm';
  modalConfirmAction.className = 'btn';
  modalConfirmAction.addEventListener('click',
closeModalHandler);

  modal.append(modalText);
  modal.append(modalCancelAction);
  modal.append(modalConfirmAction);

  document.body.append(modal);

  backdrop = document.createElement('div');
  backdrop.className = 'backdrop';

  backdrop.addEventListener('click',
closeModalHandler);

  document.body.append(backdrop);
}

function closeModalHandler() {
  modal.remove();
  modal = null;

  backdrop.remove();
  backdrop = null;
}
```

***

Приведенный выше код работает для одной конкретной кнопки на форме (при нажатии на кнопку создается модальное окно, содержащее две кнопки: «Confirm» и «Cancel»).

Если в проекте присутствует множество компонентов с похожими свойствами, то нам придется писать один и тот же код для каждого элемента. В этом случае мы получаем очень длинный и сложный в сопровождении код.

Однако, если мы перепишем приведенный выше код на *React.js*, получится что-то вроде этого:

```
***
import Todo from './components/Todo';

function App() {
  return (
    <div>
      <h1>My Todos</h1>
      <Todo text='Learn React' />
    </div>
  );
}

export default App;
                                                    ***
```

В коде выше мы видим основной компонент “App”, который рендерит один компонент “ToDo”. А код компонента “ToDo” находится в отдельном файле:

```
***
import {useState} from 'react';

import Backdrop from './Backdrop';
import Modal from './Modal';

function Todo(props) {
  const [showModal, setShowModal] = useState();

  function showModalHandler() {
```

```
    setShowModal(true);
  }

  function closeModalHandler() {
    setShowModal(false);
  }

  return (
    <div className='card'>
      <h2>{props.text}</h2>
      <div className='actions'>
        <button className='btn'
onClick={showModalHandler}>
          Delete
        </button>
      </div>
      {showModal && <Backdrop
onClick={closeModalHandler} />}
      {showModal && <Modal text='Are you sure?'
onClose={closeModalHandler} />}
    </div>
  );
}

export default Todo;
```

***

Вы можете использовать этот компонент несколько раз, просто добавив тег компонента в основной код компонента “App”:

```
***
import Todo from './components/Todo';

function App() {
  return (
    <div>
      <h1>My Todos</h1>
      <Todo text='Learn React One' />
      <Todo text='Learn React Two' />
```

```
        <Todo text='Learn React Three' />
      </div>
    );
}

export default App;
```

* * *

Как видно из приведенного выше кода, *React.js* позволяет писать декларативный, хорошо структурированный и многократно используемый код, который выполняется на стороне клиента без дополнительных запросов к серверу.

8.3. Настройка рабочего пространства

Чтобы начать работу с *React.js*, необходимо настроить рабочее пространство. Для его настройки требуется выполнить следующие действия.

1. Установите *Node.js*. Для работы *React.js* необходимо установить диспетчер пакетов *Node.js*. Для этого посетите официальный веб-сайт *Node.js*[1] и загрузите последнюю версию LTS (долгосрочная поддержка), подходящую для вашей операционной системы (рис. 7). После установки у вас будет доступ к диспетчеру пакетов *npm* (*Node Package Manager*).

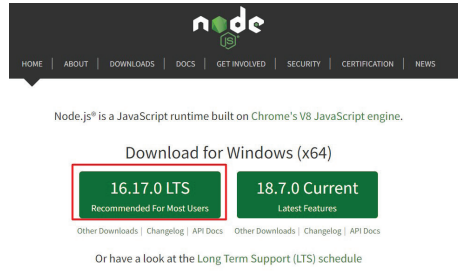


Рис. 7. Установка *Node.js*

[1] Node.js : [website]. URL: https://nodejs.org (date of accessed: 02.02.2024).

Чтобы убедиться в правильности установки “npm”, мы можем проверить установленную версию “npm” с помощью команды ниже:

```
npm -v
```

2. Создайте папку на рабочем столе.
3. Откройте командную строку из текущей папки (для этого можно просто ввести команду “cmd” в адресной строке текущего окна и нажать “Enter”).
4. Создайте новый проект *React.js*: запустите следующую команду, чтобы создать новый проект:

```
npx create-react-app my-app
```

Эта команда создает новый проект *React.js* с именем “my-app” со всеми необходимыми зависимостями и базовой структурой проекта. Обратите внимание, что во время установки проекта все необходимые зависимости загружаются через пакетный менеджер “npm”. Это может занять некоторое время.

5. Запустите сервер разработки: перейдите в папку проекта, выполнив команду:

```
cd my-app
```

и запустите приложение с помощью следующей команды:

```
npm start
```

Главная страница приложения *React.js* по умолчанию выглядит следующим образом (рис. 8).

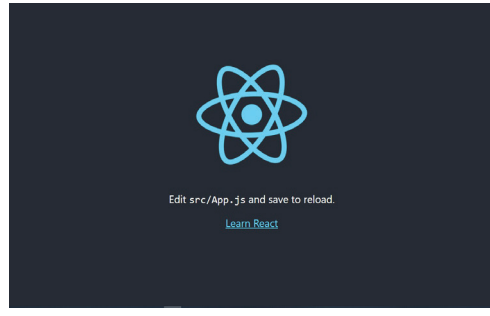


Рис. 8. Главная страница приложения *React.js*

В проекте включена функция мгновенного «горячего обновления». Это означает, что любые изменения, внесенные в код, автоматически вызовут перезагрузку страницы, что позволит вам увидеть обновления в режиме реального времени после сохранения кода.

6. Откройте проект в *Visual Studio Code* (*File -> Open Folder* -> Выберите папку проекта).

8.4. Структура проекта

CRA (команда “create-react-app”) подготавливает базовую структуру для каждого проекта (рис. 9).

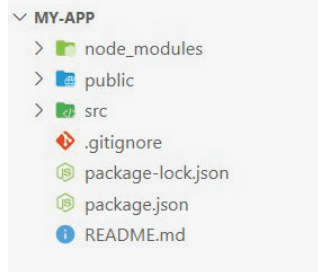


Рис. 9. Структура проекта

Перейдя в каталог проекта, вы увидите несколько каталогов:

- *node_modules* — каталог с загруженными зависимостями;
- *public* — каталог для статических файлов. На эту папку настроен сервер разработки. Она должна содержать все статические ресурсы проекта (шрифты, общие стили, HTML, изображения и т. д.);
- *src* — каталог для исходного кода проекта. CRA автоматически подготавливает несколько файлов с компонентами для демонстрационных целей. Затем их можно будет удалить. В корне проекта находятся файлы, стандартные для любого проекта: *package.json*, *package-lock.json* и *readme.md*.

8.5. Сценарии по умолчанию

В проекте, созданном с помощью CRA, в файле “package.json” доступны несколько скриптов. Этот набор может отличаться в зависимости от шаблона, который использовался при создании проекта:

- *start* — запуск проекта. По умолчанию сервер разработки подключается к порту 3000;
- *build* — подготовка сборки проекта. Он используется при подготовке проекта к отправке на сервер;
- *test* — запуск автоматизированного тестирования;
- *eject* — сброс конфигурации. Для более точной настройки.

8.6. Дополнительные настройки

Инструмент CRA настраивает все самостоятельно. Однако настройки по умолчанию не всегда могут вам подходить. Может возникнуть, например, такая проблема: порт 3000 (он используется по умолчанию) может быть занят другой службой. В качестве альтернативы вы как разработчик хотите, чтобы проект запускался в определенном браузере, а не в том, который загружается по умолчанию.

Некоторые настройки по умолчанию могут быть переопределены с помощью переменных окружения. В документации для CRA есть отдельный раздел «Расширенная конфигурация». В нем перечислены все переменные окружения, которые позволяют вам это сделать.

Вы можете установить желаемое значение для определенной переменной во время запуска проекта прямо в терминале. Команда будет выглядеть следующим образом:

```
PORT=6000 npm run start
```

Здесь мы устанавливаем новое значение для переменной “PORT”. Когда приложение запустится, оно будет загружено на порт 6000 вместо 3000 по умолчанию. Другие доступные переменные окружения могут быть установлены следующим образом:

PORT=6000 HOST=127.0.0.10 npm run start

У этого метода есть один серьезный недостаток: переменные окружения должны устанавливаться при каждом запуске приложения. Это очень трудоемко и неэффективно при использовании различных сценариев. Однако эта проблема также может быть решена.

Все переменные окружения могут быть заданы в специальном конфигурационном файле. В этом случае необходимо создать файл в корне проекта с именем ".env.local" и прописать в нем необходимые переменные окружения. При запуске проекта файл будет проанализирован, и переменные окружения будут установлены автоматически.

Файл ".env.local" не может быть передан в систему управления версиями. В разных случаях могут использоваться разные среды. Этот файл создается для конкретной задачи и используется только во время разработки. Переменные окружения могут содержать некоторую важную информацию или конфиденциальные данные, поэтому настройки указанного файла не должны попадать в *Git*. Добавьте соответствующее правило в файл ".gitignore".

Полный список поддерживаемых переменных среды приведен в документации. Наиболее часто используются следующие:

- HOST — адрес хостинга, на котором запущен проект;
- PORT — порт для запуска сервера разработки;
- HTTPS — использование HTTPS по умолчанию;
- BROWSER_ARGS — дополнительные аргументы для браузера, в котором приложение открывается после запуска;
- DISABLE_ESLINT_PLUGIN — активация/деактивация плагина "eslint-webpack-plugin".

8.7. Компоненты

Давайте теперь создадим простой компонент *React.js*, чтобы понять, как различные концепции, которые мы обсуждали, сочетаются друг с другом. Откроем созданный с помощью

CRA проект в редакторе кода и найдем в нем папку “src”. Внутри нее создадим новый файл с именем “Greeting.js”. Добавим следующий код компонента:

```
***
import React from 'react';

const Greeting = () => {
  return <h1>Hello, React!</h1>;
}

export default Greeting;
                                                    ***
```

В приведенном выше коде мы импортировали *React.js* и определили функциональный компонент под названием “Greeting.js”. Компонент возвращает код “JSX”, который отображает элемент <h1> с текстом “Hello, React!”

Чтобы использовать этот компонент, откроем “src/App.js” файл и заменим его содержимое следующим кодом:

```
***
import React from 'react';
import Greeting from './Greeting';

const App = () => {
  return <Greeting />;
}

export default App;
                                                    ***
```

В приведенном выше коде мы импортировали компонент “Greeting.js” и использовали его в компоненте “App.js”. После запуска приложения мы увидим сообщение “Hello, React!”, отображаемое в браузере.

Давайте создадим пример регистрационной формы, используя *React.js*. Мы напишем компонент формы, который будет собирать имя пользователя, адрес электронной почты и пароль.

1. Создадим новый файл с именем “RegistrationForm.js” внутри папки “src”.
2. Добавим следующий код для определения компонента регистрационной формы:

```
***
import React, {useState} from 'react';

const RegistrationForm = () => {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    // Here, you can perform validation or submit
the form data to an API
    console.log('Form submitted:', {name, email,
password});
    // Reset the form
    setName('');
    setEmail('');
    setPassword('');
  }

  return (
    <form onSubmit={handleSubmit}>
      <h2>Registration Form</h2>
      <label>
        Name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
          required
        />
      </label>
      <label>
```

```
          Email:
          <input
            type="email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
            required
          />
        </label>
        <label>
          Password:
          <input
            type="password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
            required
          />
        </label>
        <button type="submit">Register</button>
      </form>
    );
}

export default RegistrationForm;
```

***

В приведенном выше коде мы импортируем *React.js* и используем функцию (хук) “useState()” из пакета “react”. Хук “userState()” используется для управления состоянием формы. Мы объявили три переменные состояния: “name”, “email” и “password”, а также соответствующие им функции установки значений (“setName”, “setEmail”, “SetPassword”).

Функция “handleSubmit” запускается при отправке формы. Она предотвращает поведение формы по умолчанию, выводит данные формы в консоль (в этом конкретном случае) и сбрасывает поля формы, устанавливая переменные состояния как пустые строки.

Внутри инструкции “return” мы отображаем элемент <form> с полями регистрационной формы: “name”, “email” и “password”.

Каждое поле ввода связано с соответствующей переменной состояния и обработчиком события “onChange”, который обновляет состояние по мере ввода пользователем данных.

3. Чтобы использовать компонент “RegistrationForm.js”, откроем “src/App.js” файл и заменим его содержимое следующим кодом:

```
***
import React from 'react';
import RegistrationForm from './RegistrationForm';

const App = () => {
  return <RegistrationForm />;
}

export default App;
                                                    ***
```

В приведенном выше коде мы импортировали компонент “RegistrationForm.js” и отрисовали его внутри компонента “App”.

4. Запустим приложение командой “npm start” в терминале. Регистрационная форма будет отображена в окне браузера.

Если заполнить поля формы и отправить ее, в инструментах разработчика (нажмите “F12”) можно заметить, что данные формы заносятся в консоль, а поля формы очищаются.

5. Добавим немного стилей CSS для регистрационной формы. Для этого создадим отдельный CSS-файл и импортируем его в компоненты. Вот как мы можем это сделать:

- создадим новый файл с именем “RegistrationForm.css” внутри папки “src” проекта;
- добавим следующий CSS-код для оформления регистрационной формы:

```
***
.registration-form {
  max-width: 400px;
  margin: 0 auto;
  padding: 20px;
  background-color: #f4f4f4;
```

```
  border: 1px solid #ccc;
  border-radius: 5px;
}

.registration-form h2 {
  margin-top: 0;
  text-align: center;
}

.registration-form label {
  display: block;
  margin-bottom: 10px;
}

.registration-form input[type="text"],
.registration-form input[type="email"],
.registration-form input[type="password"] {
  width: 100%;
  padding: 8px;
  font-size: 16px;
  border-radius: 3px;
  border: 1px solid #ccc;
}

.registration-form button[type="submit"] {
  width: 100%;
  padding: 10px;
  font-size: 16px;
  background-color: #4caf50;
  color: white;
  border: none;
  border-radius: 3px;
  cursor: pointer;
}

.registration-form button[type="submit"]: hover {
 background-color: #45a049;
}
```

***

В приведенном выше CSS-коде мы добавили стили для класса ".registration-form", а также специальные стили для заголовков, лейблов, полей ввода и кнопки отправки.

6. Откроем "RegistrationForm.js" и импортируем CSS-файл, добавив следующую строку вверху:

```
***
import './RegistrationForm.css';
                                                    ***
```

7. Применим класс ".registration-form" к элементу <form> в функции "return" компонента "RegistrationForm.js" следующим образом:

```
***
return (
    <form className="registration-form"onSubmit=
{handleSubmit}>
      {/* Остальной код формы */}
    </form>
);
                                                    ***
```

8. Сохраним изменения. Если запустить приложение *React.js* на данном этапе, мы увидим, что регистрационная форма оформлена в соответствии со стилями CSS, которые мы определили.

С помощью стилей CSS форма регистрации будет иметь центрированный макет, светло-серый фон и рамку. Поля ввода и кнопка отправки будут иметь согласованные стили.

Вы можете изменять стили CSS в соответствии с вашими предпочтениями или добавлять дополнительные стили для дальнейшей настройки внешнего вида регистрационной формы.

8.8. Документация

Более подробную информацию по использованию инструмента CRA вы можете найти в официальной документации. Документацию можно найти на следующем ресурсе: *Create-*

React-App[1]. Интересные разделы: «Расширенное использование», «Внутренняя интеграция».

CRA — это инструмент для подготовки основы проекта с использованием *React.js*. Вы получаете готовый проект и не задумываетесь о настройке *webpack*, *babel* и других вспомогательных модулей. Все это настраивается «из коробки», и при необходимости конфигурация может быть изменена.

8.9. Пример создания простого проекта на React.js

С помощью утилиты CRA создадим новый проект с именем “my-app” и откроем его в редакторе кода *VS Code*.

После создания *React.js* проекта с помощью инструмента “create-react-app” нам необходимо удалить лишний код, который в нем присутствует по умолчанию.

Удалим из папки “src” проекта следующие файлы (рис. 10): “App.test.js”, “logo.svg”, “reportWebVitals.js”, “setupTests.js”.

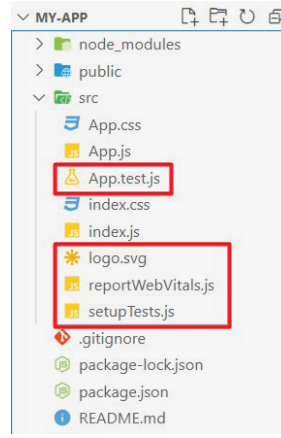


Рис. 10. Структура проекта

[1] Create React App : [website]. URL: https://create-react-app.dev (date of accessed: 02.02.2024).

Отформатируем код файла “index.js”, как показано на рис. 11.

index.js M

src > index.js > ...

```
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  import './index.css';
4  import App from './App';
5
6  const root = ReactDOM.createRoot(document.getElementById('root'));
7  root.render(
8    <React.StrictMode>
9      <App />
10   </React.StrictMode>
11 );
```

Рис. 11. Файл “index.js”

В файле “index.css” удалим весь код css стилей (рис. 12).

index.js M | index.css M

src > index.css

```
1
```

Рис. 12. Файл стилей “index.css”

Мы добавим свой собственный код в файл стилей чуть позже.

В основном файле “App.js” удалим импорты в верхней части файла, а также разметку, которая добавлена по умолчанию при создании проекта. После внесения изменений код будет выглядеть следующим образом (рис. 13).

App.js M

src > App.js > ...

```
1  import './App.css';
2
3  function App() {
4    return (
5      <div>This is empty React.js project</div>
6    );
7  }
8
9  export default App;
```

Рис. 13. Файл “App.js” после форматирования

На данном этапе мы отчистили проект от кода, который находится там по умолчанию.

В файле “package.json” содержатся необходимые зависимости, которые используются в приложении. Список зависимостей можно посмотреть в разделе “dependencies” (рис. 14).

Рис. 14. Файл “package.json”

Также здесь находится раздел “scripts”, который содержит различные сценарии для работы с проектом. Например, сценарий “start” запускает проект. При этом происходит сборка всех необходимых зависимостей и текущего кода в формат, понятный для отображения браузером.

Для проверки работы проекта можем запустить приложение на выполнение.

В терминале перейдем в директорию проекта:

cd my-app

После запустим проект на выполнение:

npm start

После запуска в браузере откроется главная страница приложения (рис. 15).

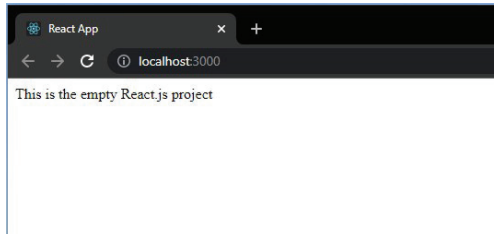


Рис. 15. Главная страница приложения в браузере

Когда приложение запущено, веб-сервер постоянно прослушивает все изменения в коде и автоматически отображает их на веб-странице в браузере. Все ошибки, которые будут происходить во время выполнения приложения, будут видны в окне браузера, а также в терминале *VS Code*.

Для того чтобы остановить выполнение приложения, необходимо в терминале ввести команду:

ctr + C

Как работает приложение *React.js*?

На данный момент в проекте есть два файла с кодом *JavaScript*: “index.js” и “App.js”. Первый является точкой входа в приложение. Код в этом файле будет исполнен первым, когда мы загрузим приложение в браузер. В верхней части файла подключаются все необходимые зависимости, которые будут использоваться в коде этого файла (рис. 16).

Например, в строке

```
***
import ReactDOM from 'react-dom/client';
                                        ***
```

из библиотеки “react-dom” импортируется объект “ReactDOM”, с которым мы далее можем работать в коде.

index.js M ×

src > index.js > ...

```
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  import './index.css';
4  import App from './App';
5
6  const root = ReactDOM.createRoot(document.getElementById('root'));
7  root.render(
8      <App />
9  );
```

Рис. 16. Файл “index.js”

Далее, вызывается метод “render” объекта “ReactDOM”, который находит в DOM веб-страницы элемент с “id="root"” и рендерит в него разметку, которая находится в файле компонента “App.js” (рис. 17). Данный компонент также импортируется в верхней части файла “index.js”.

В свою очередь, элемент “root” находится в файле “index.html” папки “public” корневого каталога приложения.

```
6  const root = ReactDOM.createRoot(document.getElementById('root'));
7  root.render(
8      <App />
9  );
```

Рис. 17. Фрагмент файла “index.js”

На рис. выше можно увидеть необычный синтаксис: код HTML внутри кода *JavaScript*. Это синтаксис “JSX”. Он позволяет использовать синтаксис HTML (теги) внутри кода приложения. Он непонятен браузеру (код конвертируется в стандартный *JavaScript* в ходе выполнения приложения), но мы можем его использовать в ходе разработки приложения.

Вся разработка приложений на *React.js* построена на создании компонентов, которые содержат разметку (HTML + JSX синтаксис), и дальнейшем связывании этих компонентов в единое целое.

Если открыть инструменты разработчика во время выполнения приложения, мы можем обнаружить корневой элемент “root”, в котором находится вся разметка, которую мы поместили в компонент “App.js” (рис. 18).

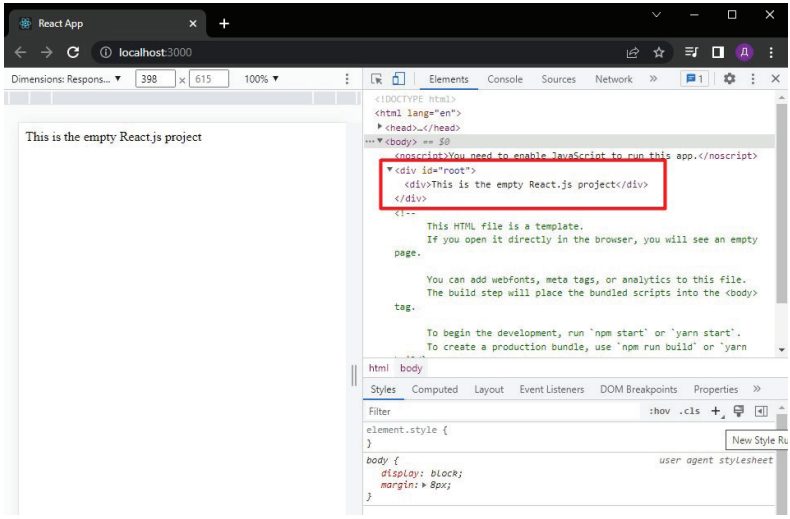


Рис. 18. Инструменты разработчика. Элемент “root”

Файл компонента “App.js” содержит стандартную функцию *JavaScript* (рис. 19). Но функция возвращает в нашем случае некоторую разметку.

Рис. 19. Код компонента “App.js”

Важно. Любой компонент внутри *React.js* приложения является обычной функцией *JavaScript*, возвращающей HTMJ+JSX разметку, которая будет отрисовываться браузером.

Создаем разметку компонента

Изменим код файла “App.js” на следующий:

```
***
function App() {
  return (
  <div>
    <h1>My Todos</h1>
    <div>
      <h2>Title</h2>
      <div>
        <button>Delete</button>
      </div>
    </div>
  </div>
);
}

export default App;
***
```

И запустим приложение в браузере:

npm start

Вывод будет следующим (рис. 20).

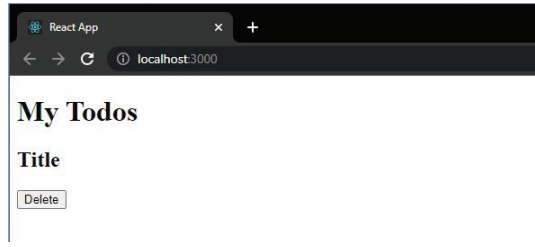


Рис. 20. Вывод главной страницы в браузере

На данный момент к кнопке не привязано никакого события, а “Title” — это просто статичный текст.

Далее, добавим код для стилизации страницы. Для этого откроем файл “index.css” и добавим в него следующий CSS-код:

```
***
* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 3rem;
  background-color: #dfdfdf;
}

h1,
h2 {
  color: #333333;
}

.btn {
  font: inherit;
  padding: 0.5rem 1.5rem;
  cursor: pointer;
  border-radius: 4px;
  background-color: #800040;
  color: white;
  border: 1px solid #800040;
  margin: 0 1rem;
}

.btn: hover {
  background-color: #9c1458;
  border-color: #9c1458;
}

.btn — alt {
  background-color: transparent;
  color: #800040;
}
```

```
.btn — alt: hover {
  background-color: #f8dae9;
}

.card {
  background-color: white;
  border-radius: 4px;
  box-shadow: 0 1px 4px rgba(0, 0, 0, 0.2);
  padding: 1rem;
  width: 20rem;
}

.actions {
  text-align: right;
}

.modal {
  box-shadow: 0 1px 4px rgba(0, 0, 0, 0.2);
  border-radius: 6px;
  background-color: white;
  padding: 1rem;
  text-align: center;
  width: 30rem;
  z-index: 10;
  position: fixed;
  top: 20vh;
  left: calc(50% — 15rem);
}

.background {
  position: fixed;
  z-index: 1;
  background-color: rgba(0, 0, 0, 0.75);
  width: 100%;
  height: 100vh;
  top: 0;
  left: 0;
}
```

***

Также необходимо добавить в разметку классы CSS, которые мы только что написали в файле “index.css”. Разметка компонента “App.js” будет выглядеть следующим образом:

```
***
function App() {
  return (
  <div>
    <h1>My Todos</h1>
    <div className="card">
      <h2>Title</h2>
      <div className="actions">
        <button className="btn">Delete</button>
      </div>
    </div>
  </div>
);
}

export default App;
                                                    ***
```

Важно. Для добавления названия классов в коде компонента используется не просто атрибут “class” как в стандартном HTML, а атрибут с именем “className”, который используется для задания классов в *JavaScript*.

На данном этапе весь код приложения находится в одном компоненте “App.js”. А что, если нам необходимо вывести на страницу не один элемент с “Card”, а сразу несколько?

В этом случае нам придется повторить код элемента “Card” несколько раз (рис. 21). Если будут внесены некоторые дополнительные изменения в данный элемент, то код придется переписывать для каждого элемента. Это очень трудозатратный вариант.

Именно для этого в проектах *React.js* используется концепция компонентов, отдельных блоков, которые потом собираются в один пользовательский интерфейс.

Рис. 21. Вывод нескольких элементов “Card”

По соглашению, все пользовательские компоненты хранятся в отдельной папке “components”. Создадим новую папку “components” в папке “src”, а внутри “components” добавим файл с названием “ToDo.js” (рис. 22).

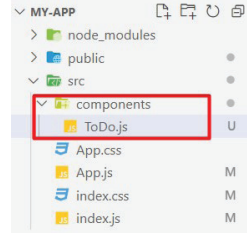


Рис. 22. Создание папки “components”
и файла “ToDo.js”

Внутри файла мы пишем следующий код:

```
***
import React from "react";

function Todo() {
  return (
    <div className="card">
      <h2>Title</h2>
      <div className="actions">
        <button className="btn">Delete</button>
      </div>
    </div>
  );
}

export default Todo;
                                                    ***
```

Для быстрой вставки каркаса функции можно использовать специальный сниппет “rfce”. Просто наберите эти символы и нажмите “Enter”.

Как уже говорилось выше, компонент — это функция *JavaScript*. В последней строке кода функция экспортируется по умолчанию для дальнейшего использования этого компонента в приложении.

> **Важно.** Название функции должно совпадать с названием файла компонента. Также название файла компонента должно начинаться с большой буквы для того, чтобы отличить компонент от названия встроенных тегов HTML.

После создания файла компонента мы можем использовать его в корневом компоненте приложения “App.js” как обычный самозакрывающийся HTML-тег (рис. 23).

Мы можем использовать этот компонент любое количество раз, просто добавляя тег компонента в разметку. При внесении

изменений в код компонента разметка всех компонентов будет изменена автоматически.

```
ToDo.js U   App.js M ×
src > App.js > ...
1  import Todo from "./components/ToDo";
2
3  function App() {
4    return (
5      <div>
6        <h1>My Todos</h1>
7        <Todo />
8        <Todo />
9        <Todo />
10     </div>
11   );
12 }
13
14 export default App;
```

Рис. 23. Компонент “App.js”
с тремя компонентами “ToDo.js”

Вывод в браузере будет следующим (рис. 24).

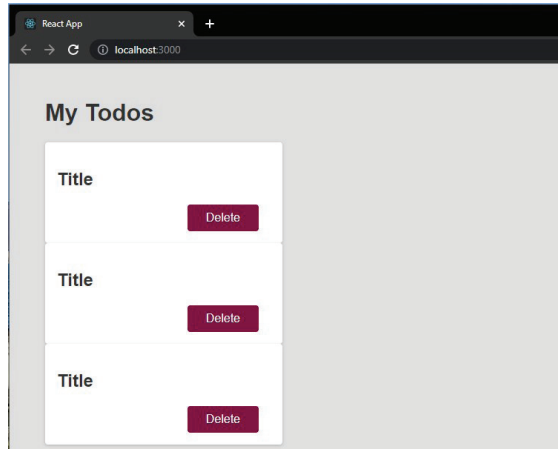


Рис. 24. Главная страница приложения в окне браузера

Далее, рассмотрим концепцию “props” для передачи параметров между компонентами.

8.10. Props

На данный момент мы разместили внутри главного компонента “App.js” три компонента “ToDo.js”. Но сейчас эти три компонента отображают одну и ту же информацию, которая жестко закодирована внутри компонента. Каким образом мы можем передавать различные данные внутрь каждого компонента и отрисовывать его в соответствии с этими данными?

Чтобы иметь возможность конфигурировать компоненты и принимать различные данные, мы используем стандартную концепцию, которая используется в стандартных функциях *JavaScript* и элементах HTML.

Добавим каждому компоненту атрибут “Title” с соответствующим значением, как показано на рис. 25.

App.js M ×

src > App.js > ...

```
1  import Todo from "./components/ToDo";
2
3  function App() {
4    return (
5      <div>
6        <h1>My Todos</h1>
7        <Todo title="one"/>
8        <Todo title="two"/>
9        <Todo title="three"/>
10     </div>
11   );
12 }
13
14 export default App;
```

Рис. 25. Добавление атрибута “Title” компонентам

Если мы сейчас запустим приложение, то этот код не даст никакого эффекта, потому что компоненты “ToDo” ничего не знают о передаче этих данных. Чтобы принимать данные внутри компонента, необходимо использовать концепцию, которая называется “props”.

В каждом компоненте мы можем принимать специальный объект, который в *React.js* обычно называют “props” (на самом деле, название можно сделать любым). Свойствами этого объек-

та являются именно те атрибуты, которые были переданы компоненту при создании его экземпляра в других компонентах.

В коде мы работаем с функциональными компонентами, которые представляют собой обычные функции *JavaScript*. Благодаря библиотеке *React.js* мы можем использовать особые конструкции. Как известно, функции могут принимать аргументы, хотя их можно использовать и без аргументов. Мы можем создать универсальную функцию, например, для сложения чисел, которая принимает два числа и возвращает их сумму:

```
***
function summOfNumbers(a, b) {
  return a + b
}
                                                    ***
```

То, что возвращает такая функция, будет зависеть от того, какие значения аргументов ей передали при вызове. При работе с компонентами *React.js* мы можем действовать подобным образом.

Для передачи параметров в компонент “ToDo” изменим его код, как показано на рис. 26.

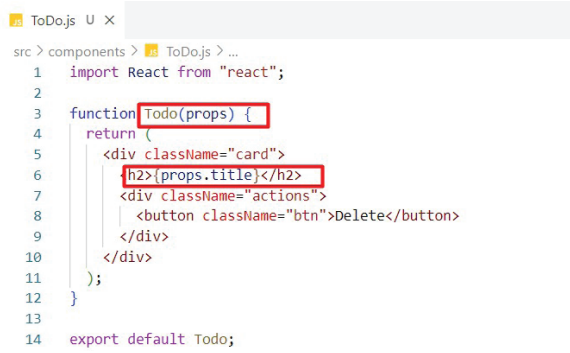


Рис. 26. Использование “props” в компоненте

В коде выше мы передали объект (“props”) в функцию компонента, который благодаря работе *React.js* уже содержит свой-

ства, которые мы передали в главном компоненте “App.js”. После передачи “props” мы можем использовать свойства этого объекта в коде компонента: мы забрали свойство “props.title” и с помощью синтаксиса JSX вставили ссылку на него в тег <h2>. Таким образом при отрисовке компонентов каждый экземпляр компонента будет получать свое собственное значение “titile” и подставлять его в разметку (рис. 27).

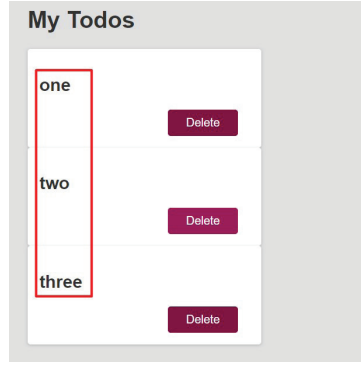


Рис. 27. Главная страница приложения
в окне браузера

Далее, рассмотрим возможности *React.js* по обработке событий.

8.11. Обработка событий

Для обработки каких-либо событий в нативном *JavaScript* обычно присутствует следующий алгоритм действий.

1. Мы ищем в коде необходимый элемент, к которому будет добавлен “EventListener”:

```
***
document.getElementById('button').addEventListener
('click');
                                                    ***
```

2. После, где-то в коде, мы добавляем к необходимому элементу соответствующий атрибут, которому присваиваем некоторую функцию:

```
***
<button onClick="clickButton()"></button>
                                                    ***
```

И только после этого при нажатии на кнопку будет происходить описанное событие.

Этот код представляет собой *императивный подход*. Мы описываем то, что должно произойти и что необходимо сделать шаг за шагом.

В *React.js* мы используем *декларативный подход* для описания целевых результатов. Для прослушивания событий мы просто добавляем дополнительный атрибут к определенному тегу. Все теги, которые мы используем в разметке, здесь являются компонентами. И мы можем добавлять к тегам специализированные атрибуты, которых нет в обычном HTML.

Например, если мы хотим добавить к кнопке прослушивание события “onClick”, то мы напишем следующий код (изменим код компонента “ToDo”):

```
***
import React from "react";

function Todo(props) {

  function deleteHendler () {
    console.log("Button clicked");
  }

  return (
    <div className="card">
      <h2>{props.title}</h2>
      <div className="actions">
        <button className="btn" onClick={deleteHendler}
>Delete</button>
```

```
        </div>
      </div>
    );
}

export default Todo;
```

***

Атрибуту “onClick” будет присвоено динамическое выражение, которое содержит название функции, определенной здесь же, в коде компонента.

> **Важно.** Значение атрибута в данном случае — это просто название функции без круглых скобок. Здесь **не нужно** исполнять функцию.

Если запустить приложение на выполнение и нажать на кнопки несколько раз, то в консоли можно увидеть результат работы функции (рис. 28).

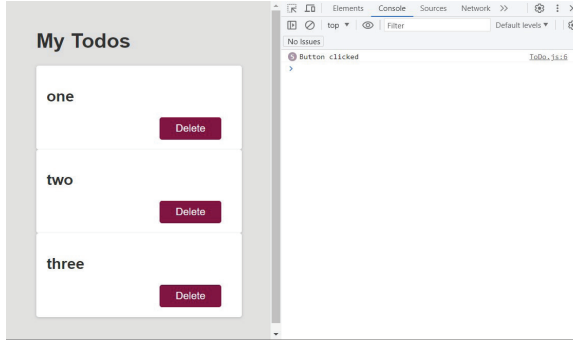


Рис. 28. Результат работы функции “deleteHandler”

Если мы добавим в код функции еще одну строку, например:

```
***
function deleteHendler () {
```

```
    console.log("Button clicked");
    console.log(props.title);
}
```

* * *

то мы также увидим, какая кнопка была нажата (рис. 29).

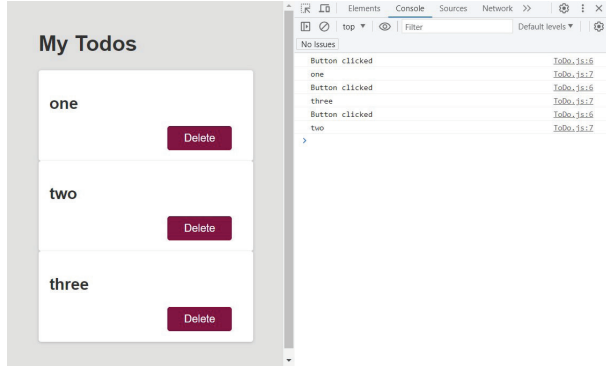


Рис. 29. Результат работы функции “deleteHandler”

Три кнопки, которые показаны на рис. выше, имеют один и тот же код, но дают разные результаты исполнения функции. В этом и состоит преимущество использования *React.js*: мы просто определяем один раз логику работы компонента (как шаблона) и можем использовать код любое количество раз, при этом получая отличные друг от друга результаты его выполнения.

8.12. Состояние приложения

Сделаем поведение приложения немного более реалистичным. При нажатии на кнопки компонентов “ToDo” будет появляться модальное окно, в котором мы сможем сделать выбор: удалять “ToDo” или нет.

Добавим к проекту новый компонент, который будет отвечать за отрисовку модального окна (рис. 30).

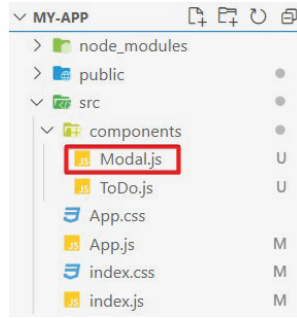


Рис. 30. Компонент “Modal.js”

Напишем код для компонента, как показано на рис. ниже (рис. 31).

Modal.js U ×

src > components > Modal.js > ...

```
1   import React from 'react'
2
3   function Modal(props) {
4     return (
5       <div className='modal'>
6           <p>Do you want to delete this ToDo?</p>
7           <button className='btn btn--alt'>Cancel</button>
8           <button className='btn'>Confirm</button>
9       </div>
10    );
11  }
12
13  export default Modal;
```

Рис. 31. Код компонента “Modal.js”

Разметка содержит один блок, внутри которого располагаются параграф и две кнопки. Название функции, как уже говорилось ранее, должно совпадать с названием файла компонента. Также в нижней части кода необходимо предусмотреть экспорт компонента (последняя строка в коде).

Для контейнера “div” добавим класс “modal”, который присутствует в файле “index.css”. Для кнопок также добавим необходимые классы для стилизации элементов.

При появлении модального окна нам необходимо немного затемнить задний фон. За это будет отвечать отдельный компонент. Создадим его в папке “components” с названием “Background.js” (рис. 32).

Background.js U ×

src > components > Background.js > ...

```
1  import React from 'react'
2
3  function Background() {
4    return (
5      <div className='background'/>
6    )
7  }
8
9  export default Background;
```

Рис. 32. Код компонента “Background.js”

В коде выше мы возвращаем разметку, в которой присутствует только один контейнер “div”. В данном случае мы можем использовать тег “div” как самозакрывающийся, т. к. он не содержит никакого контента. Эту возможность предоставляет синтаксис JSX.

Добавим созданные компоненты в основной компонент “App.js” (рис. 33).

App.js M ×

src > App.js > ...

```
1  import Todo from "./components/ToDo";
2  import Modal from "./components/Modal";
3  import Background from "./components/Background";
4
5  function App() {
6    return (
7      <div>
8        <h1>My Todos</h1>
9        <Todo title="one"/>
10       <Todo title="two"/>
11       <Todo title="three"/>
12       <Modal/>
13       <Background/>
14     </div>
15   );
16 }
17
18 export default App;
```

Рис. 33. Добавление компонентов в разметку

Если мы сохраним и запустим проект в текущем состоянии на выполнение, то получим следующий результат (рис. 34).

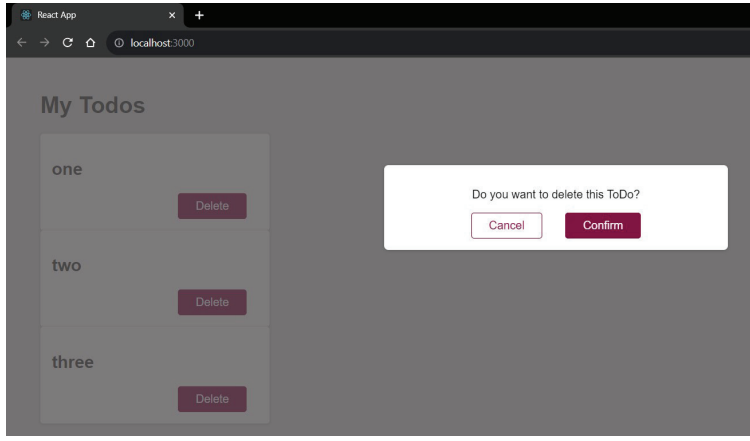


Рис. 34. Результат работы кода на данном этапе

Модальное окно и затемняющий слой будут видны постоянно, а также не будет возможности закрыть модальное окно. На данный момент мы знаем, как создавать компоненты, как их переиспользовать, как строить интерфейс из этих компонентов. Но наш интерфейс сейчас на 100 % статичный.

Для того чтобы иметь возможность каким-либо образом взаимодействовать с интерфейсом, нам необходимо изменять «Состояние приложения». На текущий момент состояние приложения предполагает появление на экране трех “ToDo”, а также постоянно видимый “Background”. Наша задача сделать так, чтобы модальное окно и затемняющий слой появлялись только при нажатии на кнопку “Delete”, т. е. менялось «Состояние приложения». Мы можем это сделать средствами встроенной в *React.js* концепции состояния приложения.

В коде компонента “ToDo” имеется функция “deleteHandler” (рис. 35), которая срабатывает, когда пользователь нажимает на кнопку “Delete”.

Рис. 35. Функция “deleteHandler” компонента “ToDo”

В момент нажатия на кнопку нам необходимо менять состояние приложения, в котором модальное окно открыто.

Для того чтобы работать с состоянием приложения в *React .js*, импортируем эту функциональность из библиотеки *React.js*:

```
***
import {useState} from "react";
                                                    ***
```

> **Важно.** В терминологии *React.js* функция “useState” — это *React Hook*. Также существуют и другие хуки: “useEffect()”, “useRef()”, “useContext()”, используемые для разных задач.

При использовании хука “useState” мы можем регистрировать различные состояния приложения, и библиотека *React.js* будет реагировать на изменения этого состояния. Хуки могут вызываться внутри функциональных компонентов.

Используем хук “useState” для управления состоянием компонента “ToDo” (рис. 36).

Хук “useState” всегда возвращает массив из двух элементов: текущее состояние компонента и функцию для изменения этого состояния.

Пояснения к рис. 36:

– *строка 5* — используется деструктурирующее присваивание, с помощью которого мы распаковываем массив в несколько переменных.

Каждый раз, когда мы меняем значение состояния компонента, *React.js* будет перерисовывать компонент в соответствии с этим состоянием;

– строка 8 — устанавливаем состояние “modalIsOpen” в значение “true”, т. е., когда функция “deleteHandler” будет вызвана, на экране появится модальное окно.

ToDo.js U ×

src > components > ToDo.js > ...

```
1  import React from "react";
2  import {useState} from "react";
3
4  function Todo(props) {
5    const [modalIsOpen, setModalIsOpen] = useState(false);
6
7    function deleteHendler () {
8      setModalIsOpen(true);
9    }
10
11   return (
12     <div className="card">
13       <h2>{props.title}</h2>
14       <div className="actions">
15         <button className="btn" onClick={deleteHendler}>Delete</button>
16       </div>
17     </div>
18   );
19 }
20
21 export default Todo;
```

Рис. 36. Использование хука “useState()” в компоненте “ToDo”

Изменим файл “App.js”. Удалим импорты компонентов, а также теги компонентов. Файл “App.js” будет выглядеть следующим образом (рис. 37).

Добавим импорты компонентов, а также теги этих компонентов в файл “ToDo.js” (рис. 38).

На данном этапе модальное окно и затемняющий фон также будут видны постоянно.

Далее, для управления состоянием модального окна и затемняющего слоя напишем следующий код (изменены 21 и 22 строки, см. рис. 39).

App.js M

src > App.js > ...

```
import Todo from "./components/ToDo";

function App() {
  return (
    <div>
      <h1>My Todos</h1>
      <Todo title="one"/>
      <Todo title="two"/>
      <Todo title="three"/>
    </div>
  );
}

export default App;
```

Рис. 37. Компонент “App.js”

ToDo.js U

src > components > ToDo.js > ...

```
import React from "react";
import { useState } from "react";
import Modal from "./Modal";
import Background from "./Background";

function Todo(props) {
  const [modalIsOpen, setModalIsOpen] = useState(false);

  function deleteHendler() {
    setModalIsOpen(true);
  }

  return (
    <div className="card">
      <h2>{props.title}</h2>
      <div className="actions">
        <button className="btn" onClick={deleteHendler}>
          Delete
        </button>
      </div>
      <Modal />
      <Background />
    </div>
  );
}

export default Todo;
```

Рис. 38. Код компонента “ToDo” на данном этапе

ToDo.js U ×

src > components > ToDo.js > ...

```
1   import React from "react";
2   import { useState } from "react";
3   import Modal from "./Modal";
4   import Background from "./Background";
5
6   function Todo(props) {
7     const [modalIsOpen, setModalIsOpen] = useState(false);
8
9     function deleteHendler() {
10      setModalIsOpen(true);
11    }
12
13    return (
14      <div className="card">
15        <h2>{props.title}</h2>
16        <div className="actions">
17          <button className="btn" onClick={deleteHendler}>
18            Delete
19          </button>
20        </div>
21        {modalIsOpen ? <Modal /> : null}
22        {modalIsOpen ? <Background /> : null}
23      </div>
24    );
25  }
26
27  export default Todo;
```

Рис. 39. Измененный код компонента “ToDo”

Если запустить приложение на данном этапе, то мы уже не увидим модальное окно при загрузке страницы. Оно будет появляться только при нажатии на кнопку “Delete”.

Однако сейчас мы не можем взаимодействовать с модальным окном. Также у нас нет возможности закрыть это модальное окно.

Для того чтобы мы могли закрыть модальное окно при нажатии на затемняющий слой (“Background”), добавим еще одну функцию в файл “ToDo.js”. И передадим эту функцию в компонент “Background” как значение атрибута “onCancel” (см. рис. 40, строки 13–15 и 26).

ToDo.js U ×

src > components > ToDo.js > ...

```
1  import React from "react";
2  import { useState } from "react";
3  import Modal from "./Modal";
4  import Background from "./Background";
5
6  function Todo(props) {
7    const [modalIsOpen, setModalIsOpen] = useState(false);
8
9    function deleteHendler() {
10     setModalIsOpen(true);
11   }
12
13 function closeModalHandler() {
14   setModalIsOpen(false);
15 }
16
17   return (
18     <div className="card">
19       <h2>{props.title}</h2>
20       <div className="actions">
21         <button className="btn" onClick={deleteHendler}>
22           Delete
23         </button>
24       </div>
25       {modalIsOpen ? <Modal /> : null}
26       {modalIsOpen ? <Background oncCancel={closeModalHandler}/> : null}
27     </div>
28   );
29 }
30
31 export default Todo;
```

Рис. 40. Передача функции как значение атрибута “onCancel”

На 26 строке мы передали функцию как значение в компонент “Background”. Но сам компонент пока не может получить эти параметры, поскольку не знает об их существовании. Компоненты, которые мы пишем сами, не имеют возможности использовать события в качестве параметров по умолчанию. Нам необходимо вручную передавать параметры компоненту (например, встроенный компонент “button” уже имеет атрибуты по умолчанию).

Для того чтобы компонент принял эти параметры, изменим код файла “Background.js”, как показано ниже (рис. 41).

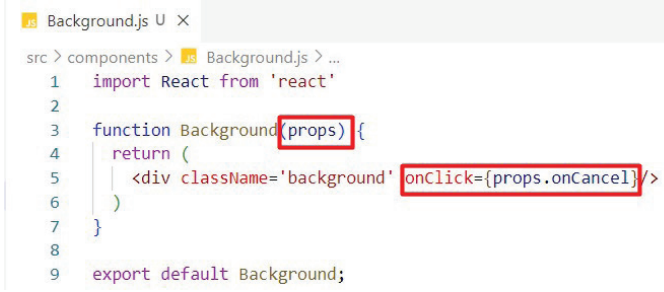


Рис. 41. Код компонента “Background”

В данном случае мы передаем из компонента “ToDo” через атрибут “onCancel” функцию “closeModalHandler” как значение атрибута, которая будет срабатывать, когда пользователь кликнет на любом месте затемняющего слоя.

Добавим подобный код для работы модального окна. Когда пользователь нажимает на кнопки “Cancel” или “Confirm”, модальное окно будет закрываться (не будем добавлять какой-либо дополнительной логики для кнопки “Confirm”, она также будет закрывать модальное окно).

Для этого передадим функцию “closeModalhandler” как значение через атрибуты “onCancel” и “onConfirm” (поскольку этот компонент написан нами лично, мы можем передавать любые имена атрибутов) (рис. 42, строка 25).

ToDo.js U

src > components > ToDo.js > ...

```
1  import React from "react";
2  import { useState } from "react";
3  import Modal from "./Modal";
4  import Background from "./Background";
5
6  function Todo(props) {
7    const [modalIsOpen, setModalIsOpen] = useState(false);
8
9    function deleteHendler() {
10     setModalIsOpen(true);
11   }
12
13 function closeModalHandler() {
14   setModalIsOpen(false);
15 }
16
17   return (
18     <div className="card">
19       <h2>{props.title}</h2>
20       <div className="actions">
21         <button className="btn" onClick={deleteHendler}>
22           Delete
23         </button>
24       </div>
25       {modalIsOpen ? <Modal onCancel={closeModalHandler} onConfirm={closeModalHandler}/> : null}
26       {modalIsOpen ? <Background oncCancel={closeModalHandler}/> : null}
27     </div>
28   );
29 }
30
31 export default Todo;
```

Рис. 42. Передача функции “closeModalHandler” как значения атрибутов “onCancel” и “onConfirm”

Далее, принимаем эти параметры через “props” и передаем в атрибуты “onClick” для элементов “button” в компоненте “Modal” (рис. 43, строки 7 и 8).

Modal.js U ×

src > components > Modal.js > Modal

```
1  import React from 'react'
2
3  function Modal(props) {
4    return (
5      <div className='modal'>
6          <p>Do you want to delete this ToDo?</p>
7          <button className='btn btn--alt' onClick={props.onCancel}>Cancel</button>
8          <button className='btn' onClick={props.onConfirm}>Confirm</button>
9      </div>
10   );
11 }
12
13 export default Modal;
```

Рис. 43. Итоговый код компонента “Modal”

Сейчас при нажатии на кнопки модального окна (“Modal”) или на любом месте затемняющего слоя (“Background”) модальное окно будет закрываться, т. е. происходит изменение состояния компонентов, и *React.js* мгновенно перерисовывает интерфейс.

8.13. React Hooks: useState(), useEffect()

В начале развития библиотеки *React.js* ее основу составляли классы. Соответственно, все программирование на *React.js* было построено на написании классов. Со временем в библиотеку вносились изменения. Сегодня на смену классам пришли так называемые функциональные компоненты. По сути, это обычные функции *JavaScript*, которые возвращают некоторую разметку. Также проекты на функциональных компонентах менее тяжелые, т. к. здесь отсутствует какое-либо наследование от базовых сущностей. Код довольно гибкий и хорошо масштабируемый.

Программирование с использованием классов предполагало некоторый жизненный цикл приложения, который можно было

использовать для отслеживания состояния компонентов во времени. В настоящее время, после перехода к программированию на функциональных компонентах, эта концепция эволюционировала в концепцию *React Hooks*. Хуки — это по факту обычные функции *JavaScript*, которые позволяют использовать все особенности и возможности *React.js* вместе с синтаксисом JSX.

Hooks — это функции, которые позволяют «подключаться» к возможностям отслеживания состояния и жизненного цикла функциональных компонентов.

Hooks не работают внутри классов — они позволяют вам использовать *React.js* без классов.

Далее, рассмотрим примеры работы *React Hooks*.

useState()

Для демонстрации работы хука “useState” создадим новый проект с помощью инструмента “create-react-app”.

Очистим проект от лишнего кода. В итоге получим следующую структуру (рис. 44).

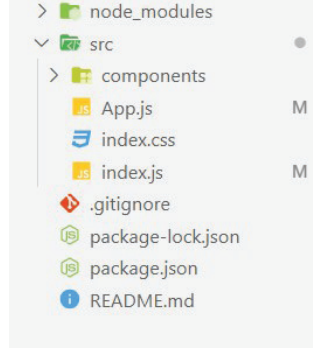


Рис. 44. Структура проекта

Для минимальной стилизации компонентов добавим в файл “index.html” (находится в папке *public*) ссылку на фреймворк *Bootstrap* (рис. 45).

index.html M

public > index.html > ...

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css/bootstrap.min.css" rel="stylesheet"
    integrity="sha384-4bw+/aepP/YC94hEpVNVgiZdgIC5+VKNBQNGCHeKRQN+PtmoHDEXuppvnDJzQIu9" crossorigin="anonymous">
  <title>React App</title>
</head>

<body>
  <div id="root"></div>
</body>

</html>
```

Рис. 45. Код файла “index.html”

Ссылку на фреймворк *Bootstrap* можно найти по следующему адресу: https://getbootstrap.com/docs/5.2/getting-started/introduction/.

Весь код файла “index.css” заменим на следующий:

```
***
* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 3rem;
  background-color: #dfdfdf;
}

h1, h2 {
  color: #333333;
}
```

***

Добавим немного кода в файл “App.js”. Это будет счетчик (h1), а также две кнопки для увеличения и уменьшения значений счетчика (рис. 46).

В браузере приложение будет выглядеть как на рис. 47.

App.js M ×

src > App.js > ...

```
1  import React from "react";
2
3  function App() {
4    return (
5      <div>
6        <h1>Counter</h1>
7        <button className="btn btn-primary">+</button>
8        <button className="btn btn-danger">-</button>
9      </div>
10   );
11 }
12
13 export default App;
```

Рис. 46. Код компонента “App.js”

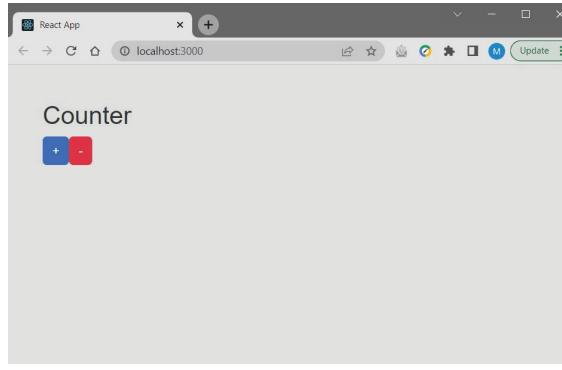


Рис. 47. Главная страница в окне браузера

Для работы с счетчиком нам необходимо подключить *React Hook* “useState()”. Для этого скорректируем импорт в верхней части файла “App.js”.

```
***
import {React, useState} from "react";
                                    ***
```

Хук “useState()” является функцией, поэтому мы можем передать в нее начальное состояние компонента. В нашем случае это будет “0”.

Данная функция всегда возвращает массив из двух элементов: текущее состояние компонента, а также функцию для изменения этого состояния. Можно убедиться в этом, если добавить еще две строки кода в компонент “App.js” (рис. 48).

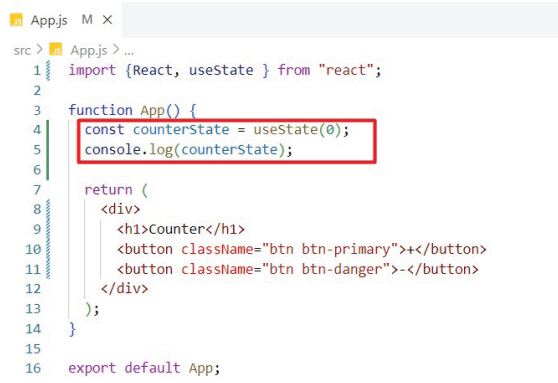


Рис. 48. Измененный файл “App.js”

Если открыть инструменты разработчика, то будет видно, что это действительно массив, который содержит некоторое значение и функцию (рис. 49).

Рис. 49. Инструменты разработчика, вывод в консоли

Удалим две добавленные строки кода (4 и 5 на рис. выше), они были нужны только для демонстрации.

Добавим код для работы с состоянием счетчика (рис. 50).

App.js M ×

src > App.js > ...

```
1  import { React, useState } from "react";
2
3  function App() {
4    const [counter, setCounter] = useState(0);
5
6    function increment() {
7      setCounter(counter + 1);
8    }
9
10   function decrement() {
11     setCounter(counter - 1);
12   }
13
14   return (
15     <div>
16       <h1>Counter: {counter}</h1>
17       <button onClick={increment} className="btn btn-primary">+</button>
18       <button onClick={decrement} className="btn btn-danger">-</button>
19     </div>
20   );
21 }
22
23 export default App;
```

Рис. 50. Код компонента “App.js”

Пояснение к рис. 50:

– строка 4 — используется деструктурирующее присваивание, с помощью которого мы распаковываем массив в несколько переменных для упрощения работы с состоянием;
– строки 6–8 — пишем функцию “increment”, которая меняет состояние переменной “counter” на 1 вверх;
– строки 10–12 — пишем функцию “decrement”, которая меняет состояние переменной “counter” на 1 вниз;
– строка 16 — в теге заголовка “h1” добавляем динамическую ссылку на переменную “counter”;
– строки 17–18 — для двух кнопок добавляем атрибуты “onClick”, которым передаем функции как значение.

Важно. *React Hooks* нельзя использовать внутри условных операторов, а также различных видов циклов.

После запуска проекта мы увидим следующий интерфейс (рис. 51).

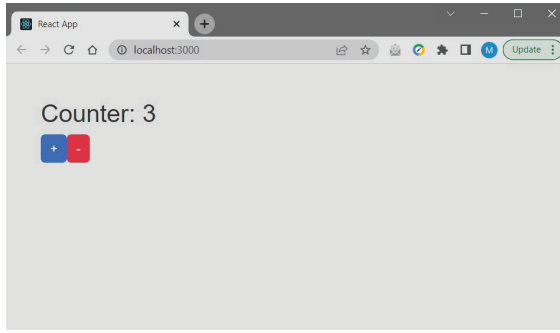


Рис. 51. Главная страница в окне браузера

Если покликать на кнопки, то счетчик будет менять свое значение, т. е. состояние счетчика меняется при нажатии на кнопки. *React.js* отслеживает это изменение и перерисовывает компонент, но уже с другими значениями переменных.

Здесь необходимо пояснить, что хук “useState” работает асинхронно, и значение переменной, которое мы отслеживаем, будет перезаписано только после перерисовки компонента. Например, если мы захотим увеличить счетчик сразу на две единицы и напишем такой код:

```
***
function increment() {
  setCounter(counter + 1);
  setCounter(counter + 1);
}
                                                    ***
```

то значение счетчика все равно будет увеличиваться только на единицу.

Эту проблему можно решить, используя другой синтаксис, который позволяет использовать предыдущее значение отслеживаемой переменной.

Перепишем код функции “increment”, как показано ниже:

```
***
function increment() {
  setCounter((prevCounter) => {
    return prevCounter + 1;
  })
  //This is other variant of the code above
  setCounter(prev => prev + 1);
}
                                                    ***
```

Как видно из кода выше, мы можем использовать хук “useState” любое количество раз в одном и том же компоненте.

Представленный код функции “increment” позволяет увеличивать значение счетчика на 2 единицы, основываясь на точном значении предыдущего состояния (изменение состояния через *callback*).

useEffect()

Очистим код компонента “App.js” и напишем код компонента “App.js”, основываясь на полученных знаниях о хуке “useState”:

```
***
import {React, useState, useEffect} from "react";

function App() {
  const [type, setType] = useState("posts");

  return (
    <>
      <div>
        <button className="btn" onClick={() =>
setType('users')}>Users</button>
        <button className="btn" onClick={() =>
setType('todos')}>ToDo</button>
        <button className="btn" onClick={() =>
setType('posts')}>Posts</button>
```

```
        </div>
        <hr/>
        <h1>{type}</h1>
      </>
    );
}

export default App;
```

* * *

Когда мы вызываем хук “useEffect”, мы как бы говорим *React.js* запустить нашу функцию “Effect” после перерисовки и изменений в DOM, т. е. мы хотим произвести какие-либо дополнительные действия, но уже после рендеринга компонента.

Хуки “useEffect” объявляются внутри компонента, поэтому у них есть доступ к его свойствам и состоянию. По умолчанию *React.js* запускает функции “Effect” после каждого рендеринга, включая первый.

Синтаксис хука “useEffect” выглядит следующим образом (показан в рамке на рис. 52).

Рис. 52. Синтаксис хука “useEffect”

В браузере видно (рис. 53), что если мы нажимаем на кнопки, то компонент ре-рендерится (перерисовывается), и каждый раз срабатывает хук “useEffect”, которому в данном случае мы передаем callback-функцию вывода состояния переменной “type” в консоль.

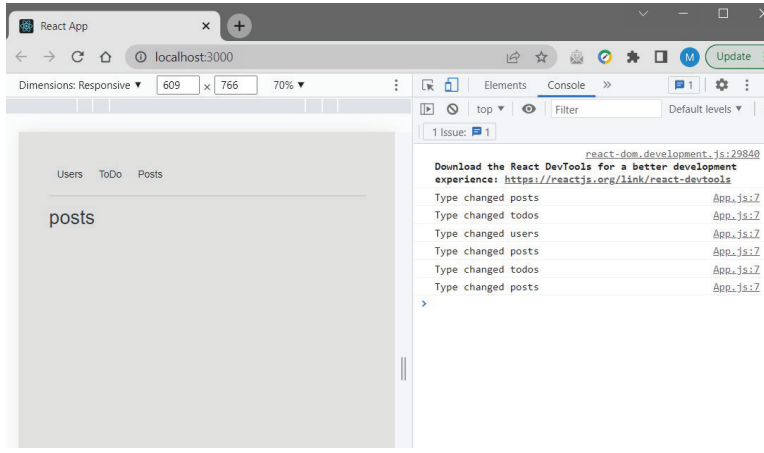


Рис. 53. Инструменты разработчика. Вывод в консоль

Если мы будем нажимать на одну и ту же кнопку (например, на кнопку “Posts”) несколько раз подряд, то в консоли не будут выводится новые строки. Это происходит из-за того, что в синтаксисе “useEffect” также присутствует специальная переменная, называемая “dependencies” (рис. 53). Она содержит массив переменных, которые служат триггерами для исполнения функции “useEffect”. Если состояние этих переменных не меняется, то и функция “useEffect” не будет срабатывать.

Изменим код, который передаем в хук “useEffect”. Внутри него мы будем получать данные с удаленного сервера, используя бесплатный сервис *JSONPlaceholder*[1]. После внесения изменений код будет выглядеть следующим образом (рис. 54).

[1] JSONPlaceholder : [website]. URL: http://jsonplaceholder.typicode.com (date of accessed: 02.02.2024).

App.js M ×

src > App.js > ...

```
1  import { React, useState, useEffect } from "react";
2
3  function App() {
4    const [type, setType] = useState("posts");
5    const [data, setData] = useState([]);
6
7    useEffect(() => {
8      fetch(`https://jsonplaceholder.typicode.com/${type}`)
9        .then((response) => response.json())
10       .then((json) => setData(json));
11   }, [type]);
12
13   return (
14     <>
15       <div>
16         <button className="btn" onClick={() => setType("users")}>Users</button>
17         <button className="btn" onClick={() => setType("todos")}>ToDo</button>
18         <button className="btn" onClick={() => setType("posts")}>Posts</button>
19       </div>
20       <hr />
21       <pre>{JSON.stringify(data, null, 2)}</pre>
22     </>
23   );
24 }
25
26 export default App;
```

Рис. 54. Измененный код компонента “App.js”

Пояснение к рис. 54:

– строка 5 — добавляем еще один хук “useState” для отслеживания состояния данных, полученных от сервера (переменная “data”);
– строки 8–10 — получаем данные с удаленного сервера с помощью синтаксиса “Promise” (документацию можно посмотреть здесь: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise);
– строка 8 — строка запроса, которая передается методу “fetch”, формируется с помощью синтаксиса «Шаблонизации строк» (ставим обратные кавычки, в конце строки ставим знак “$” и после него в фигурных скобках указываем на динамическую переменную “type”. Документация: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals);
– строка 10 — меняем состояние переменной “data”, добавляя в нее полученные данные;

– строка 21 — выводим на страницу минимально отформатированные данные, полученные с сервера (используется встроенный JS объект JSON: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON).

В браузере можно увидеть, что все необходимые данные уже появляются на экране (рис. 55).

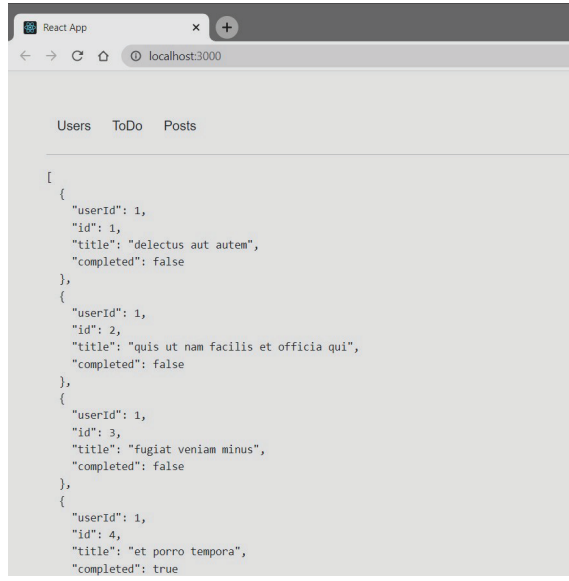


Рис. 55. Главная страница в браузере

Если понажимать на кнопки “Users”, “ToDo” или “Posts”, новые данные будут подтягиваться с сервера и отображаться на странице, т. е. на каждое изменение состояния (при нажатии кнопок) мы делаем асинхронный запрос и получаем новый набор данных (срабатывает хук “useEffect”). При этом мы отслеживаем переменную “type” и при ее изменении получаем некоторые “side effects” (побочные эффекты). Работу кода можно увидеть в инструментах разработчика на вкладке “Network” (рис. 56).

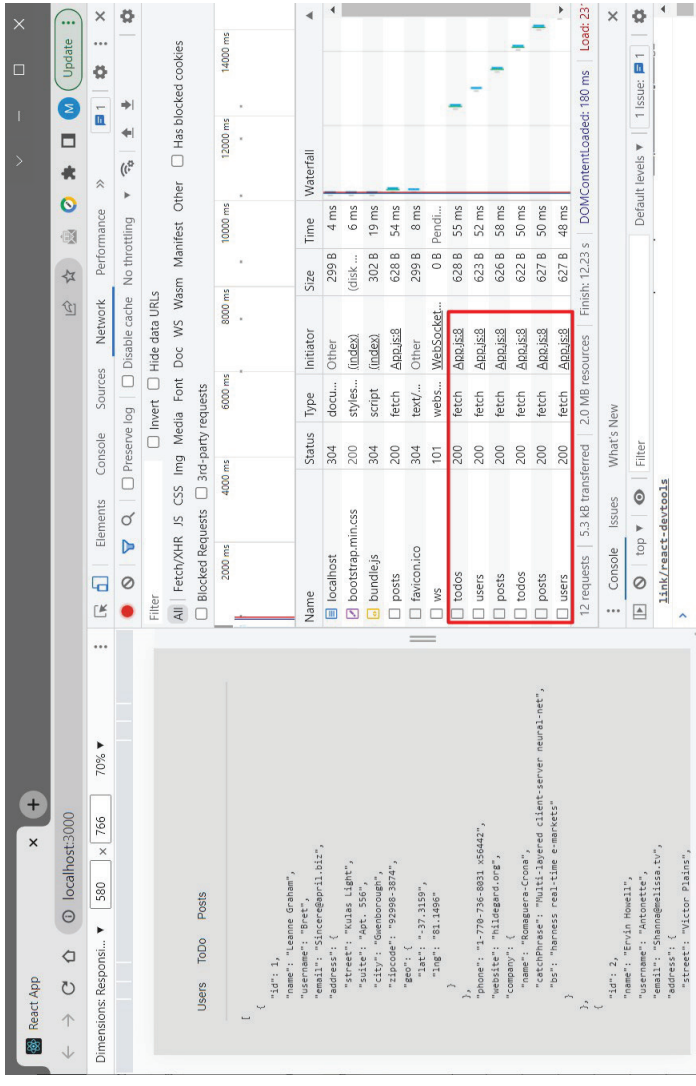


Рис. 56. Инструменты разработчика. Вкладка “Network”

Выше мы рассмотрели только один пример использования хука “useEffect”. Он также может использоваться для эмуляции работы с жизненным циклом компонента, т. е. можно отследить, когда компонент полностью отрендерится (отресуется) и будет готов к работе.

Более подробное описание с примерами можно посмотреть в официальной документации[1].

8.14. Маршрутизация в приложении. React-Router

Создадим новую папку.

Откроем командную строку из созданной папки.

Создадим с помощью инструмента “create-react-app” новый проект *React.js* с именем “upcoming-events”. Для этого введем в консоли команду:

npx create-react-app upcoming-events

После создания проекта *React.js* нам необходимо удалить лишний код, который в нем присутствует по умолчанию.

Удалим из папки “src” проекта следующие файлы: “App.test.js”, “logo.svg”, “reportWebVitals.js”, “setupTests.js” (рис. 57).

Отформатируем код файла “index.js”, как показано ниже:

```
***
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.
getElementById('root'));
root.render(
  <React.StrictMode>
```

[1] Hooks at a Glance : [website]. URL: https://reactjs.org/docs/hooks-overview.html (date of accessed: 02.02.2024).

```
    <App />
  </React.StrictMode>
);
```

***

Из файла “index.css” удалим весь код стилей CSS. В файле стилей мы добавим свой собственный код, но чуть позже.

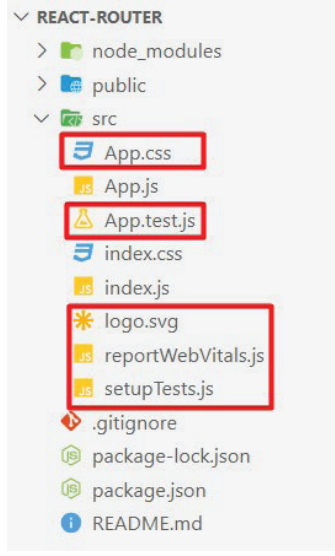


Рис. 57. Удаление лишнего кода из проекта

В основном файле “App.js” удалим импорты в верхней части файла, а также разметку, которая добавлена по умолчанию при создании проекта. После внесения изменений код будет выглядеть следующим образом:

```
***
import React from 'react';

function App() {
  return (
    <div>This is the empty React.js project</div>
```

```
  );
}

export default App;
```

***

На данном этапе мы отчистили проект от кода по умолчанию.

Запустим проект из терминала (см. команду ниже) для проверки работоспособности:

npm start

Добавим начальные стили CSS в “index.css” файл:

***

```
@import url('https://fonts.
googleapis.com/css2?family=Open+Sans:
wght@400;700&display=swap');

* {
  box-sizing: border-box;
}

body {
  font-family: 'Open Sans', sans-serif;
  margin: 0;
  background-color: rgb(217, 217, 218);
}

h1 {
  font-size: 2.5rem;
  color: #2c292b;
}

h2 {
  font-size: 1.5rem;
  color: #2c292b;
}
```

***

Когда мы работаем с приложениями SPA (*Single Page Application*), при нажатии на ссылки создается впечатление, что идет перенаправление на другие страницы и подгружаются дополнительные данные с сервера. На самом деле никаких запросов на сервер в это время обычно не происходит. Приложение постоянно находится на той же странице, которая была загружена в браузер при первом запросе.

При нажатии на ссылки мы как бы даем пользователю некоторую иллюзию наличия навигации в приложении и предоставляем библиотеке *React.js* управление контентом, который должен быть выведен на экран пользователю в текущий момент.

Преимущество данного подхода состоит в том, что нет необходимости ждать загрузки новой страницы с сервера. При этом приложение остается таким же быстрым и реактивным, потому что в этот момент оно управляется кодом *JavaScript* на стороне клиента.

React-Router

Для организации подобного поведения и эмуляции навигации в приложениях *React.js* в проект необходимо добавить специальный инструмент "React-Router". Он следит за изменениями URL внутри приложения и затем меняет контент страницы на тот, который должен быть выведен в данный момент в соответствии с текущим URL.

Документацию можно посмотреть по следующей ссылке: https://reactrouter.com/en/main/start/overview.

Добавим в проект новый модуль (зависимость), который позволяет использовать маршрутизацию в приложении без отправки запросов на сервер (т. е. управлять запросами и в соответствии с ними показывать необходимый контент). Для этого в терминале запустим команду:

npm install react-router-dom

После установки пакета в файле "package.json" мы увидим новую зависимость (рис. 58).

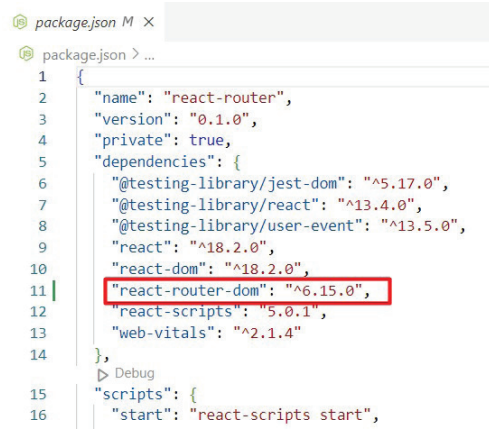


Рис. 58. Новая зависимость в файле “package.json”

После установки пакета мы можем его использовать в коде приложения.

Для удобства работы с компонентами, которые будут представлять собой страницы приложения, создадим в папке “src” новую папку “pages”.

Создавать отдельную папку необязательно. Это обычно делают для лучшей дифференциации собственных компонентов (которые встраиваются в другие компоненты) от компонентов, которые будут загружаться как страницы приложения.

В папке “pages” создадим несколько компонентов, которые будут загружаться посредством “React-Router”, когда пользователь будет посещать соответствующий URL (рис. 59).

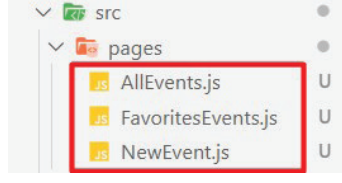


Рис. 59. Новые компоненты
в папке “pages”

Добавим код для каждого компонента, как показано ниже:

– “AllEvents.js”:

```
***
import React from 'react';

function AllEvents() {
  return (
    <div>AllEvents</div>
  );
}

export default AllEvents;
                                                    ***
```

– “FavoritesEvents.js”:

```
***
import React from 'react';

function FavoritesEvents() {
  return (
    <div>FavoritesEvents</div>
  );
}

export default FavoritesEvents;
                                                    ***
```

– “NewEvent.js”:

```
***
import React from 'react';

function NewEvent() {
  return (
    <div>NewEvent</div>
  );
}

export default NewEvent;
                                                    ***
```

На данный момент у нас есть три компонента, представляющих собой страницы приложения. Далее, необходимо использовать “React-Router”, чтобы определить, в какой момент необходимо загружать эти компоненты. Для этого перейдем в файл “index.js”. Добавим в верхней части файла еще один импорт:

```
***
import {BrowserRouter} from "react-router-dom";
                                                ***
```

и обернем тег компонента <App/> специальным тегом компонента <BrowserRouter> из пакета “React-Router” (рис. 60).

Рис. 60. Тег “BrouserRouter” в коде компонента “index.js”

В коде выше мы инициализировали пакет “React-Router”, т. е. теперь он будет следить за изменением URL.

Следующим шагом необходимо определить, какие маршруты будут поддерживаться в проекте, и какие компоненты будут при этом загружаться.

Перейдем в файл “App.js”. Здесь мы используем еще два компонента из пакета “React-Router”: “Routes” и “Route”.

Эти компоненты используются для определения маршрутов, а также конкретных компонентов, которые при этом должны быть загружены.

Добавим импорты страниц, а также импорт компонентов <Routes> и <Route>:

```
***
import React from "react";
import {Routes, Route} from 'react-router-dom';
import AllEvents from "./pages/AllEvents";
import NewEvent from "./pages/NewEvent";
import FavoritesEvents from "./pages/
FavoritesEvents";
                                                    ***
```

Определим следующую разметку для компонента “App”:

```
***
function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<AllEvents/>}>
        </Route>
        <Route path="/new-event" element=
{<NewEvent/>}>
        </Route>
        <Route path="/favorites" element=
{<FavoritesEvents/>}>
        </Route>
      </Routes>
    </div>
  );
}

export default App;
                                                    ***
```

В коде выше определены три маршрута. И сейчас приложение будет загружать указанные компоненты при соответствии текущего URL значению атрибута “path”.

Компонент “Route” содержит атрибут “path”, который получает строку — часть пути в URL приложения (после названия домена), например:

localhost:3000/allevents

В примере выше основной домен — “localhost:3000”, “path” — “/allevents”.

Также для атрибута “path” может быть определено значение по умолчанию: “path="/"”. Этот путь ведет на стартовую страницу приложения.

Сохраним код и запустим проект (рис. 61).

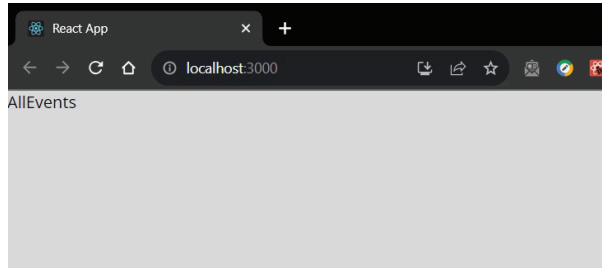


Рис. 61. Главная страница приложения в браузере

Если мы будем вводить в поисковой строке браузера адреса: *localhost:3000/*; *localhost:3000/new-event*; *localhost:3000/favorites*, то “React-Router” будет выводить на страницу соответствующие компоненты в зависимости от различных URL.

Navigation, Links

На данный момент наш проект не совсем похож на реальный. В реальности в каждом проекте для перехода между страницами существует что-то вроде навигации, меню (*Navigation Bar*).

Перейдем в папку “components” и создадим внутри новую папку “layout”. Внутри папки “layout” создадим еще один компонент “MainNav.js”, который будет содержать код навигационного меню.

Добавим следующий код в компонент “MainNav.js”:

```
***
import React from "react";

function MainNav() {
```

```
  return (
    <header>
      <div>Events</div>
      <nav>
        <ul>
          <li><a href="#">Link</a></li>
        </ul>
      </nav>
    </header>
  );
}

export default MainNav;
```

***

Как видно из кода выше, мы используем обычные ссылки для перехода между страницами. У этого подхода есть один большой недостаток: при нажатии на ссылку мы посылаем запрос на сервер, получаем ответ и перерисовываем страницу. Однако наша задача, наоборот, состоит в том, чтобы уменьшить количество запросов на сервер.

Для этой цели в *React.js* используется еще один компонент, который называется “Link”. Перепишем код компонента, как показано ниже:

```
***
import React from "react";
import {Link} from "react-router-dom";

function MainNav() {
  return (
    <header>
      <div>Upcoming Events</div>
      <nav>
        <ul>
          <li>
            <Link to="/">Events</Link>
          </li>
```

```
            <li>
              <Link to="/new-event">New Event</Link>
            </li>
            <li>
              <Link to="/favorites">Favorites</Link>
            </li>
          </ul>
        </nav>
      </header>
    );
}

export default MainNav;
```

***

В приведенном выше коде мы используем компоненты “Link” с атрибутом “to”, который принимает в качестве значения адрес URL связанного компонента.

Добавим в код основного компонента “App.js” только что созданный компонент “MainNav.js”:

```
***
import React from "react";
import {Routes, Route} from 'react-router-dom';
import AllEvents from "./pages/AllEvents";
import NewEvent from "./pages/NewEvent";
import FavoritesEvents from "./pages/
FavoritesEvents";
import MainNav from "./components/MainNav";

function App() {
  return (
    <div>
      <MainNav/>
      <Routes>
        <Route path="/" element={<AllEvents/>}>
        </Route>
        <Route path="/new-event" element={<NewEvent/>}>
        </Route>
```

```
          <Route path="/favorites" element=
{<FavoritesEvents/>}>
          </Route>
        </Routes>
      </div>
    );
}

export default App;
```

* * *

Запустим проект. В браузере мы увидим следующий вывод (рис. 62).

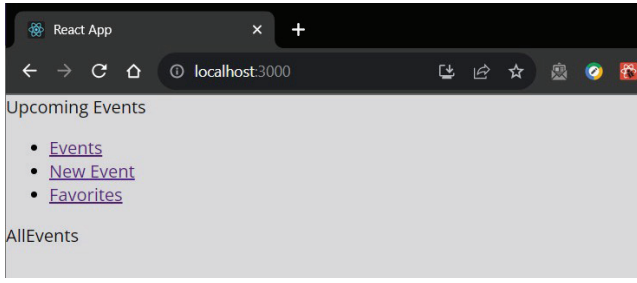


Рис. 62. Главная страница приложения в браузере

Если понажимать на ссылки, то можно увидеть, что компоненты, соответствующие ссылкам, отображаются ниже при нажатии на них.

8.15. CSS Modules

На данный момент мы почти не использовали никаких стилей для разметки. Добавим CSS-код, однако, мы не будем просто писать весь CSS-код в одном файле “index.css”, мы воспользуемся концепцией «CSS-модулей».

Часто в больших проектах для различных компонентов создаются отдельные CSS-файлы. При подключении конкретно-

го файла стилей к компоненту эти стили никак не повлияют на стилизацию других компонентов. Для проектов, которые содержат десятки или даже сотни компонентов — это крайне важно.

Создадим отдельный CSS-файл для компонента “MainNav.js”. Логично расположить файл стилей в той же папке, где расположен сам компонент. При именовании файлов необходимо придерживаться определенного подхода: название файла стилей должно содержать слово “module” (рис. 63).

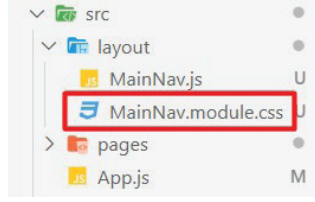


Рис. 63. Файл стилей
для компонента “MainNav.js”

Добавим в созданный файл стилей следующий код:

```
***
.header {
  width: 100%;
  height: 5rem;
  display: flex;
  align-items: center;
  justify-content: space-between;
  background-color: #f37a4a;
  padding: 0 10%;
}

.logo {
  font-size: 1.4rem;
  color: white;
  font-weight: bold;
}
```

```
.header ul {
  list-style: none;
  margin: 0;
  padding: 0;
  display: flex;
  align-items: baseline;
}

.header li {
  margin-left: 3rem;
}

.header a {
  text-decoration: none;
  font-size: 1.2rem;
  color: #f07d44;
}

.header a: hover,
.header a: active,
.header a.active {
  color: white;
}

.badge {
  background-color: #3043ef;
  color: white;
  border-radius: 12px;
  padding: 0 1rem;
  margin-left: 0.5rem;
}
```

***

Далее, мы можем импортировать классы из файла стилей в код компонента “MainNav.js” *как один объект*, который содержит *все классы* как свойства этого объекта (рис. 64).

В окне браузера можно увидеть, что все классы применились, и страница сейчас выглядит следующим образом (рис. 65).

MainNav.js U

src > layout > MainNav.js > ...

```
1  import React from "react";
2  import { Link } from "react-router-dom";
3  import classes from "./MainNav.module.css";
4
5  function MainNav() {
6    return (
7      <header className={classes.header}>
8        <div className={classes.logo}>Upcoming Events</div>
9        <nav>
10         <ul>
11           <li>
12             <Link to="/">Events</Link>
13           </li>
14           <li>
15             <Link to="/new-event">New Event</Link>
16           </li>
17           <li>
18             <Link to="/favorites">Favorites</Link>
19           </li>
20         </ul>
21       </nav>
22     </header>
23   );
24 }
25
26 export default MainNav;
```

Рис. 64. Код компонента “MainNav.js” с внесенными изменениями

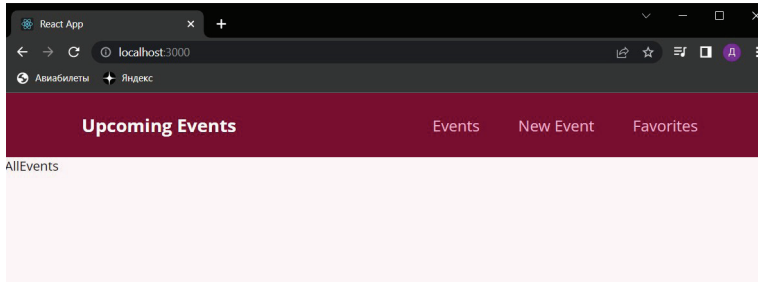


Рис. 65. Главная страница в окне браузера

Далее, поработаем со страницей “AllEvents”. Здесь нам необходимо выводить список событий, которые будут храниться в каком-то массиве.

В реальном проекте данные грузятся из удаленной базы данных. В нашем случае мы добавим некоторые статические данные.

Откроем файл “AllEvents.js” и добавим в него следующий массив статических данных (сразу после строк с импортом):

```
***
const DATA = [
  {
    id: '1',
    title: 'First Event',
    image: 'https://upload.wikimedia.org/wikipedia/
commons/thumb/c/cb/Classical_spectacular10.
jpg/1280px-Classical_spectacular10.jpg',
    eventadress: 'Mira 19',
    description: 'Some text for the first event.'
  },
  {
    id: '2',
    title: 'Second Event',
    image: 'https://upload.wikimedia.org/wikipedia/
commons/c/ca/Bernardo_Strozzi_Kontsert.jpg',
    eventadress: 'Mira 19',
    description: 'Some text for the second event.'
  }
]
                                                  ***
```

Далее, в этом же файле выведем эти данные на страницу. Ниже показан код компонента:

```
***
function AllEvents() {
  return (
    <section>
      <h1>AllEvents</h1>
        <ul>
          {DATA.map((data) => {
            return (
              <li key={data.id}>{data.title}</li>
```

```
            )
          })}
        </ul>
      </section>
    )
}

export default AllEvents
```

* * *

В коде выше мы использовали метод “map” для перебора всех элементов массива, который присутствует в синтаксисе *JavaScript*. Этот метод позволяет вызвать некоторую функцию (в нашем случае это стрелочная функция) для каждого элемента массива. Как результат, мы получаем новый массив уже с трансформированными данными, которые выводятся на страницу. На данный момент главная страница приложения выглядит следующим образом (рис. 66).

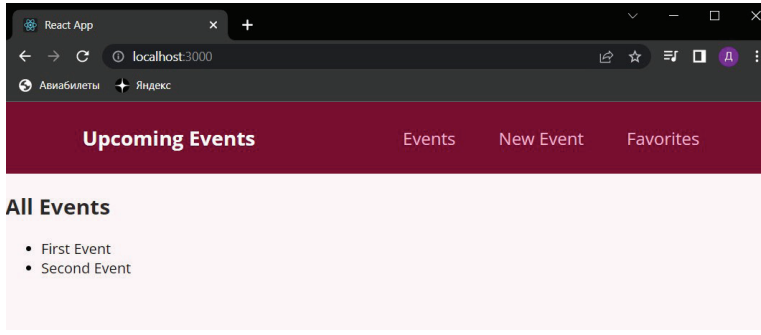


Рис. 66. Главная страница в окне браузера

На странице “AllEvents” нам необходимо вывести список предстоящих событий. Каждое событие представляет собой элемент массива, который мы получаем с удаленного сервера. Создадим дополнительные компоненты, которые будут отвечать за отрисовку каждого отдельного элемента списка (“EventItem”), а также списка целиком (“EventList”).

Добавим новую папку “events” внутри папки “components”. В папке “events” создадим файлы компонентов и файлы стилей для этих компонентов (рис. 67).

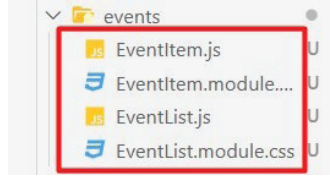


Рис. 67. Новая папка “events”
и два новых компонента с файлами стилей

Код файла “EventItem.js”:

```
***
import React from 'react'
import classes from './EventItem.module.css';

function EventItem(props) {
  return (
    <li className={classes.item}>
      <div className={classes.image}>
        <img src={props.image} alt={props.title}/>
      </div>
      <div className={classes.content}>
        <h3>{props.title}</h3>
        <address>{props.eventadress}</address>
        <p>{props.description}</p>
      </div>
      <div className={classes.actions}>
        <button>Favorites +</button>
      </div>
    </li>
  )
}

export default EventItem
                                                    ***
```

Код файла стилей “EventItem.module.css”:

```
***
.item {
  margin: 1rem 0;
}

.image {
  width: 100%;
  height: 20rem;
  overflow: hidden;
  border-top-right-radius: 6px;
  border-top-left-radius: 6px;
}

.image img {
  width: 100%;
  object-fit: cover;
}

.content {
  text-align: center;
  padding: 1rem;
}

.content h3 {
  font-size: 1.25rem;
  color: #2c292b;
}

.actions {
  padding: 1.5rem;
  text-align: center;
}

.actions button {
  font: inherit;
  cursor: pointer;
```

```
  color: #77002e;
  border: 1px solid #77002e;
  background-color: transparent;
  padding: 0.5rem 1.5rem;
  border-radius: 4px;
}

.actions button: hover,
.actions button: active {
  background-color: #ffe2ed;
}
```

***

Код файла “EventList.js”:

```
***
import React from 'react'
import classes from './EventList.module.css';
import EventItem from './EventItem.js';

function EventList(props) {
  return (
    <ul className={classes.list}>
        {props.data.map((data) => (
          <EventItem
          key={data.id}
          id={data.id}
          image={data.image}
          title={data.title}
          adress={data.adress}
          description={data.description}
          />
        ))
      }
    </ul>
  )
}

export default EventList
```

***

В коде выше мы создаем список из элементов статического массива (который мы определили в файле “AllEvents.js”) и передаем в каждый элемент необходимые свойства (“props”).

Также мы могли бы передать весь массив пропсов сразу. В этом случае нам пришлось бы деструктурировать полученный массив внутри компонента “EventItem.js”. Вы можете действовать любым из этих способов.

Код файла стилей “EventList.module.js”:

```
***
.list {
  list-style: none;
  margin: 0;
  padding: 0;
  width: 70%;
  margin: 0 auto;
}
                                                    ***
```

Далее, используем созданные компоненты. Перейдем в файл “AllEvents.js”, добавим импорт компонента:

```
***
import EventList from '../events/EventList';
                                                    ***
```

А также изменим код функции “AllEvents”:

```
***
function AllEvents() {
  return (
    <section>
      <h1>AllEvents</h1>
        <EventList data={DATA}/>
    </section>
  )
}
                                                    ***
```

В коде выше мы передаем в компонент данные из статического массива, т. к. они необходимы компоненту “EventItem”, находящемуся внутри компонента “EventList”.

В браузере мы увидим следующий вывод (рис. 68).

Стилизация приложения еще не закончена, но данные уже выводятся на страницу.

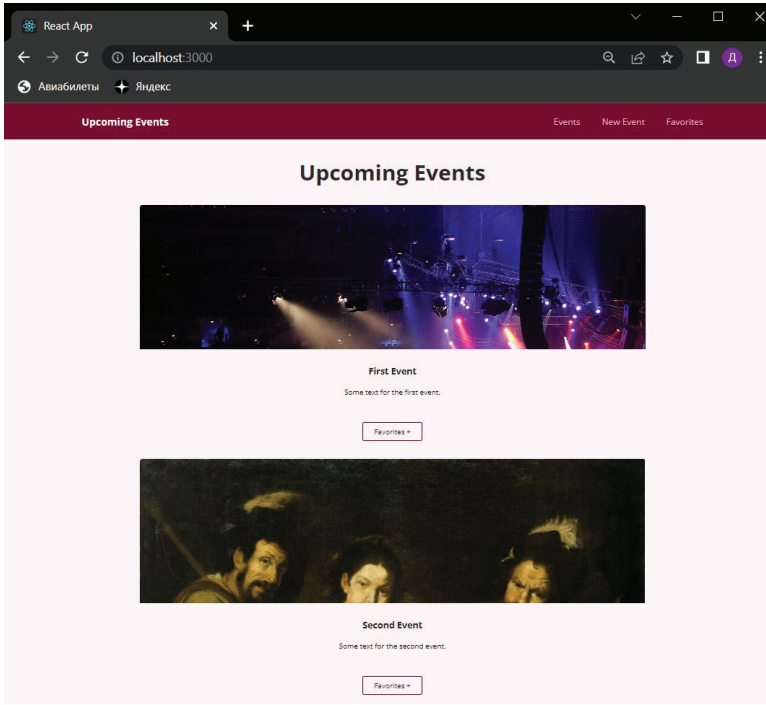


Рис. 68. Главная страница в окне браузера

В проекте выше мы рассмотрели только базовые основы веб-разработки на *React.js*. Данная библиотека представляет собой инструмент, который позволяет применять ключевые концепции, такие как компоненты, состояние (*state*), виртуальный DOM и многое другое, для создания современных веб-приложений. Эти концепции стали основой веб-разработки и помогают создавать легко поддерживаемый и масштабируемый код.

Разработка на *React.js* — это непрерывный процесс обучения и роста. Новые возможности и инструменты появляют-

ся регулярно, и вам необходимо постоянно обновлять и расширять свои знания. Ниже представлены ссылки на ресурсы в Сети, которые могут вам в этом помочь.

1. *Официальная документация React.js*[1] — этот ресурс предоставляет обширную и актуальную информацию о *React.js* и его основных концепциях.
2. *React Community*[2] — здесь вы можете найти форумы, сообщества и многое другое для общения с другими разработчиками и получения помощи.
3. *React на GitHub*[3] — исходный код *React.js*, доступен на *GitHub*. Это отличный способ изучить внутреннее устройство библиотеки.

Вопросы для самопроверки

1. Что такое *React.js*, и какова его цель в веб-разработке?
2. Как настроить базовую среду разработки *React.js*?
3. Что такое компонент *React.js*, и как его создать?
4. Что такое JSX (*JavaScript XML*), и зачем его используют в разработке *React.js*? Приведите пример кода JSX.
5. Объясните концепцию “props” в *React.js*. Как передавать данные из родительского компонента в дочерний с помощью “props”?
6. Что такое состояние (*state*) в *React.js*, и как управлять состоянием компонента?
7. Объясните назначение хуков “useState()” и “useEffect()” в функциональных компонентах *React.js*. Приведите примеры их использования.

[1] React.js : [website]. URL: https://reactjs.org/docs/getting-started.html (date of accessed: 02.02.2024).

[2] React Community : [website]. URL: https://reactjs.org/community/support.html (date of accessed: 02.02.2024).

[3] React на GitHub : [website]. URL: https://github.com/facebook/react (date of accessed: 02.02.2024).

8. Опишите концепцию обработки событий в *React.js* (например, кликов по кнопке). Приведите примеры использования обработчиков событий.
9. Обсудите роль *React Router* в создании одностраничных приложений (SPA) с *React.js*.
10. Что такое виртуальное DOM, и как *React.js* использует его для оптимизации производительности?

Заключение

Данное пособие содержит базовые знания о различных аспектах разработки веб-приложений и их влиянии на современный мир бизнеса. В заключении мы хотим подчеркнуть основные моменты, которые были изучены.

Вы познакомились с этапами, необходимыми для создания успешных веб-проектов. От начального планирования и проектирования до верстки, тестирования и развертывания — каждый этап играет важную роль в достижении целей любого веб-проекта.

Навыки работы с редакторами кода позволяют быстро создавать и редактировать HTML, CSS и *JavaScript*, сэкономив много времени и ресурсов.

Язык разметки HTML дает возможность создавать веб-страницы, определять структуру документа, создавать ссылки, изображения, списки и другие элементы, необходимые для представления информации на веб-странице.

Навыки работы с CSS позволяют стилизовать веб-страницы, делать их более привлекательными и удобными для пользователя, а также управлять позиционированием элементов и их внешним видом.

Основы *JavaScript* позволяют создавать динамические и интерактивные веб-приложения, обрабатывать события и управлять данными, включая переменные, функции, события и обработку данных.

Знакомство с библиотекой *React.js* позволяет получить представление о современных подходах к созданию мощных и масштабируемых веб-приложений. Понимание компонентного подхода и виртуального DOM дает возможность использовать

инструменты для создания сложных пользовательских интерфейсов.

Используя эти знания и навыки, руководители и специалисты всех уровней будут готовы принять вызовы современного бизнеса, создавая веб-приложения, которые соответствуют ожиданиям пользователей и помогают в достижении бизнес-целей. Эти умения также позволят оставаться конкурентоспособными в быстроразвивающемся мире веб-разработки.

Не забывайте, что веб-технологии постоянно развиваются, и важно продолжать обучение и исследование новых инструментов и методов. Желаем вам успехов во всех ваших будущих проектах и исследованиях в области веб-технологий!

Список рекомендуемой литературы

Создание технического задания

Как составить грамотное ТЗ на разработку сайта // Хабр : [сайт]. — URL: https://habr.com/ru/articles/593661/ (дата обращения: 02.02.2024).

Хортон, А. Создание технических заданий для веб-проектов / А. Хортон. — Москва : ДМК Пресс, 2023. — 200 с. — ISBN 978-5-89818-504-6.

Прототипирование

A Comprehensive Guide to UX Design Prototyping // Smashing Magazine : [website]. — URL: https://www.smashingmagazine.com/2010/06/comprehensive-guide-to-ux-design-prototyping/ (date of accessed: 02.02.2024).

Figma : [website]. — URL: https://www.figma.com/ (date of accessed: 02.02.2024).

Tondreau, B. Prototyping for Designers: Developing the Best Digital and Physical Products / B. Tondreau. — New York : O'Reilly Media, 2017. — 370 p. — ISBN 978-1491954084.

Git

Chacon, S. Pro Git / S. Chacon, B. Straub. — New York : Apress, 2021. — 814 p. — ISBN 978-1-4842-5762-9.

Documentation // Git : [website]. — URL: https://git-scm.com/doc (date of accessed: 02.02.2024).

Git Tutorial // w3schools.com : [website]. — URL: https://www.w3schools.com/git/default.asp (date of accessed: 02.02.2024).

HTML

Duckett, J. HTML and CSS: Design and Build Websites / J. Duckett. — Indianapolis : Wiley, 2011. — 512 p. — ISBN 978-1-118-00818-8.

HTML: HyperText Markup Language // MDN Web Docs : [website]. — URL: https://developer.mozilla.org/en-US/docs/Web/HTML (date of accessed: 02.02.2024).

Learn HTML // Codecademy : [website]. — URL: https://www.codecademy.com/learn/learn-html (date of accessed: 02.02.2024).

CSS

CSS: Cascading Style Sheets // MDN Web Docs : [website]. — URL: https://developer.mozilla.org/en-US/docs/Web/CSS (date of accessed: 02.02.2024).

Learn CSS // Codecademy : [website]. — URL: https://www.codecademy.com/learn/learn-css (date of accessed: 02.02.2024).

McFarland, D. CSS: The Missing Manual / D. McFarland. — Sebastopol : O'Reilly Media, 2015. — 650 p. — ISBN 978-1-491-91805-6.

JavaScript

Flanagan, D. JavaScript: The Definitive Guide / D. Flanagan. — Sebastopol : O'Reilly Media, 2020. — 706 p. — ISBN 978-1-491-95202-8.

JavaScript // MDN Web Docs : [website]. — URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (date of accessed: 02.02.2024).

Haverbeke, M. Eloquent JavaScript / M. Haverbeke. — 4th ed. — San Francisco : No Starch Press, 2018. — 472 p. — ISBN: 978-1-593-27950-9. — URL: https://eloquentJavaScript.net/ (date of accessed: 02.02.2024).

Learn JavaScript // Codecademy : [website]. — URL: https://www.codecademy.com/learn/introduction-to-JavaScript (date of accessed: 02.02.2024).

Bootstrap

Bootstrap : [website]. — URL: https://getbootstrap.com/docs/5.0/getting-started/introduction/ (date of accessed: 02.02.2024).

Bootstrap 5 Tutorial // w3schools.com : [website]. — URL: https://www.w3schools.com/bootstrap5/index.php (date of accessed: 02.02.2024).

Learn Bootstrap // Codecademy : [website]. — URL: https://www.codecademy.com/learn/learn-bootstrap (date of accessed: 02.02.2024).

React.js

Хортон, А. Разработка веб-приложений в ReactJS / А. Хортон, Р. Вайс. — Москва : ДМК Пресс, 2023. — 255 с. — ISBN 978-5-89818-503-9.

Academind // YouTube : [website]. — URL: https://www.youtube.com/@academind (date of accessed: 02.02.2024).

Learn React // Codecademy : [website]. — URL: https://www.codecademy.com/learn/react-101 (date of accessed: 02.02.2024).

React.js : [website]. — URL: https://reactjs.org/docs/getting-started.html (date of accessed: 02.02.2024).

Учебное издание

Медведев Максим Александрович
Медведева Марина Александровна

ВЕБ-ТЕХНОЛОГИИ В БИЗНЕСЕ

Редактор *Н. Ф. Тофан*
Верстка *Ю. В. Ершовой*

Подписано в печать 19.09.2024. Формат 60×84 1/16.
Бумага офсетная. Цифровая печать. Усл. печ. л. 9,8.
Уч.-изд. л. 9,0. Тираж 30 экз. Заказ № 101.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ ЦСД
620062, Екатеринбург, ул. С. Ковалевской, 5.
Тел.: +7 (343) 375-48-25, 375-46-85, 374-19-41
E-mail: rio@urfu.ru

Отпечатано в ИПЦ УрФУ ЦСД
620083, Екатеринбург, ул. Тургенева, 4.
Тел.: +7 (343) 358-93-06, 350-58-20, 350-90-13
http://print/urfu.ru


ИНДЕКС УНИВЕРСИТЕТСКОГО КНИГОИЗДАНИЯ
УНИВЕРСИТЕТСКАЯ КНИГА
2024


9 785799 639051