

# Навигация

## Введение

1. Как задавать вопросы
2. Как проходить курс
3. Горячие клавиши VS Code
4. Основы терминала

## Повторение JS

5. Чит лист JS
6. Оператор Switch
7. Методы массива
8. Метод Sort
9. Методы объекта
10. Методы строк
11. Методы чисел
12. Методы даты
13. Деструктуризация

14. Асинхронность
15. Замыкания
16. Памятка по формам

## Введение в React

17. Как создать проект Vite
18. Функция createElement
19. JSX
20. JSX + CSS
21. JSX Памятка
22. Пути в проекта Vite
23. Что такое Props
24. Метод тар в компонентах
25. Деструктуризация props
26. Оператор && в React
27. Тернарный оператор ?:
28. Условный рендеринг

## Углубление в основы React

29. Обработка событий

30. useState hook
31. useState + callback
32. Когда использовать useState
33. Декларативный vs императивный
34. Array.from() создание массива

## Важные возможности Props

35. children prop
36. prop без значения
37. Component Composition
38. Убираем prop drilling
39. Явная передача компонента

## UseEffect хук

40. Введение в useEffect
41. useEffect dependency array
42. Side effects
43. useEffect или Event для side effects
44. Классовые vs функциональные компоненты

REACT JS

# Навигация

## Необходимая теория

- 45. Классовые vs функциональные компоненты
- 46. Components, instances, elements
- 47. Как работает rendering
- 48. Что такое Виртуальный DOM
- 49. Fiber tree
- 50. Fiber tree vs Виртуальный DOM
- 51. Схема рендеринга
- 52. Мемоизация компонента
- 53. Фаза коммита
- 54. Diffing
- 55. Пакетное обновление состояний
- 56. Монтирование и размонитрование
- 57. Framework vs library

## useRef & Кастомный хук

- 58. useRef hook
- 59. Кастомный хук

## React Router

- 60. Библиотека React Router
- 61. Компонент <Link>
- 62. Компонент <Outlet />
- 63. Атрибут index
- 64. useParams() hook
- 65. useSearchParams() hook
- 66. useLocation() hook
- 67. useNavigate() hook
- 68. Компонент <Navigate />
- 69. Варианты создания ссылки

## CSS Modules

- 70. CSS Modules

## React Router 6.4+

### (С погрузкой данных)

- 71. Rout до версии 6.4
- 72. Router v6.4+ loading
- 73. Router v6.4+ errorElement
- 74. useNavigation hook
- 75. Router v6.4+ action

## Context API

- 75. Context API

## useReducer hook

- 76. useReducer()
- 77. useReducer() payload
- 78. useReducer() initialState

REACT JS

# Навигация

## **Redux + RTK**

- 79. Redux введение
- 80. Redux Thunk
- 81. Redux Toolkit (RTK)
- 82. Redux DevTools Extention
- 83. Toolkit createAsyncThunk
- 84. Обработка ошибок (RTK)
- 85. Доп. возможности AsyncThunk

## **Оптимизация**

### **производительности**

- 86. useMemo hook
- 87. useCallback hook
- 88. Boundle & LazyLoad

## **Деплой проекта**

- 89. Хэширование изображений
- 90. NPM Run Build
- 91. Размещение проекта на хостинге

# Ваши вопросы

— и как их задавать

## Отвечаю на вопросы Каждый день

Я отвечаю на вопросы каждый день,  
только на платформе, где вы проходите  
обучение.

Пожалуйста, не пишите мне в Instagram  
или Telegram.

Задавать вопросы вы можете в комментариях под видео или  
в чате на платформе.

Смотря, что есть на платформе.

## Вопрос подробно

Пожалуйста, задавайте ваши вопросы максимально подробно и развернуто

- Не вставляйте ваш код в текст сообщения.
- Не задавайте вопросы типа «не работает, что делать?» без подробных пояснений ваших действий и того, что именно не работает.
- Не пишите «не понимаю» без подробных пояснений того, что именно вы не понимаете.
- Не присылайте обрезанные скриншоты.  
(Я приближу нужное место)
- Не задавайте длинных запутанных или философских вопросов.  
Пары строк всегда достаточно.

# Не вставляйте ваш код в текст сообщения.

Пожалуйста, познакомьтесь с GitHub или, по крайней мере, с Google Disk, чтобы предоставить мне ссылку на ваш проект целиком.

Не забудьте удалить из вашего проекта ненужные элементы. Например, папку node\_modules, которая может занимать более 200 МБ.

clip-path можно ли установить иконку svg за приделами изображения clip-path? чтобы иконку было видно. Спасибо  
например иконку сейчас вижу

```
1 | видно иконку
2 |
3 | *, ::after, ::before
4 |
5 |   margin: 0
6 |   padding: 0
7 |   box-sizing: border-box
8 |
9 |
10| body
11|
12|
13|   font-family: "Roboto", sans-serif
14|   font-weight: 400
15|   color: #777777
16|   font-size: 1.6rem
17|   letter-spacing: .2rem
18|   padding: 5rem
19|   background-color: white
20|
21|
22|.container
23|
24|   background-color: #f7f7f7
25|
26|
27|.header
28|
29|
30|   position: relative
31|   height: 130vh
32|   background: linear-gradient(90deg, rgba(186, 133, 84, 0.7) 0.03%,
33|   rgba(1, 1, 1, 0.7) 99.94%), url('../img/bmw1.jpg') center / cover no-repeat
34|   clip-path: polygon(50% 0%, 83% 12%, 100% 43%, 94% 78%, 68% 100%, 32%
35|   100%, 6% 78%, 0% 43%, 17% 12%)
```

## REACT JS

# Не задавайте длинных или философских вопросов.

### Трекинг

Интересно, что текст над заголовком ( тот который понадобилось делать в верхнем регистре) и текст абзаца имеют одинаковый кегль, при этом первый понадобилось растягивать на 5%. Понятно, что с практикой придет понимание, где и на сколько нужно этот трекинг делать. Если бы можно было как-то "формулировать" рекомендацию по трекингу... Кстати, что касается интерлиньяжа, то действия в практике позволили набросать по нескольким точкам (кегль; интерлиньяж) почти линейный график зависимости одного от другого - это здорово помогает.

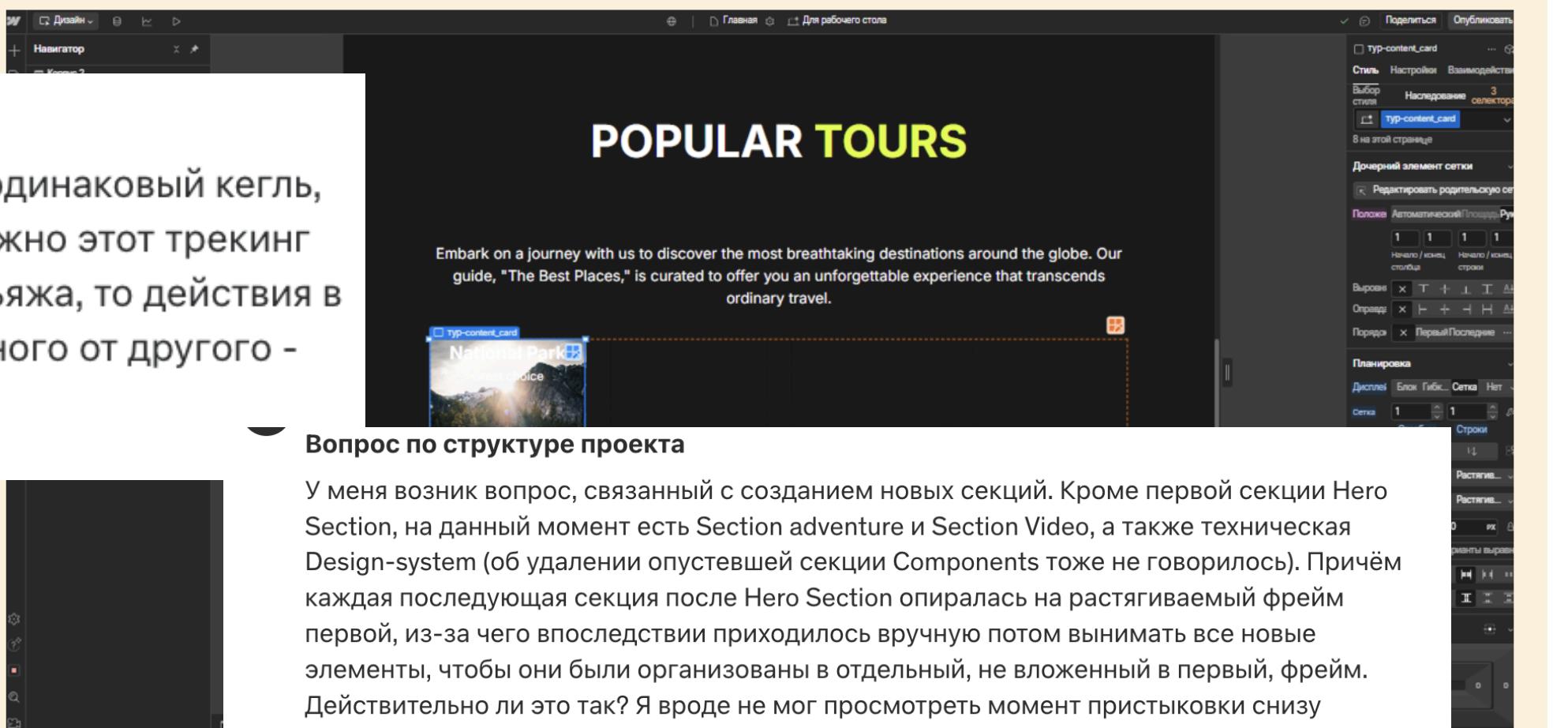
Фактически мой вопрос связан с тем, что задавал пользователь Anna 5 месяцев назад. Примерно на 7 минуте видео вы создаёте первый фиолетовый квадрат произвольного размера под отступ в 4 px, а затем, выбрав квадрат и текст внутри, делаете Auto layout с нужными падингами. У вас в видео это автоматически приводит к сжатию квадрата. У меня так не произошло. Сравнивая видео со своим экраном, я понял, что каким-то образом у вас выставилась регулировка фрейма Horizontal/Vertical resizing в положение Hug вместо Fixed width, что отразилось в подписи на голубом фоне под фреймом (раньше стояли конкретные размеры, теперь Hug\*Hug). Но вот когда и почему это произошло непонятно. Нужны пояснения.

сделала все как вы говорили но когда удалила 3 карточки и заменила их на первую переделанную вот что выдает и никак не хочет вебфлоу их видеть при этом они есть - они как бы есть но их как бы нету ,посоветуйте как исправить пожалуйста , немогу дальше иди по курсу из за этого ,я так думаю что вебфлоу всетаки "косячная" ну неможет быть так что все идет ровно по вашему под копирку и запара вот так вот , причем откатывалась и начинала заново делать и снова на этом моменте запара, я подозреваю что Вебфлоу неможет справляться с кучей див блоками в гриде когда там очень все глубоко друг в друге по куча раз , слишком много косяков идет, а именно отличий интерфейса и работы настроек самих, заметила пока прохожу ваше обучение точнее пытаюсь как то пройти - к примеру с кнопками мне сам вебфлоу кнопку просто переделал визуал когда наводишь и никак не исправляется ободок , даже заного пересоздала клонку и визуал при наведении - всеравно рамку после перезахода в вебфлоу она сама все исправляет без каких либо наследований и как нерабочий ( закрытый) интерфейс настроек становится , вот теперь такая фигня началась когда куча див блоков внутри кучи гридов (на фото) и тоже пересобираю с нуля и опять такая фигня выдается причем все четко под ваше видео делаю по милли секунде пересматриваю и перепроверяю .

P.S. не смотрите что все по русски просто перед тем как сделать скрин нечаянно нажала на яндекс перевод страницы на русский.

Короче говоря вебфлоу живет своей жизнью и не хочет чтоб верстали с кучей гридов и див блоками , а хочет только чтоб тупо текст был ,картиночки ,но уже с наложенным текстом и без исправления на нем текстов иначе закапризничает и нафиг все снесет . И желательно чтоб пользователи пользовались только уже готовыми контейнерами и блоками с уже за ранее заданными параметрами и количеством функционала. И в вебфлоу уже за ранее заданы свои параметры и ограничения которые визуально видны и их не обойти так как это в систему сайта вшито , и скажем курс который выложен 9 - 12 мес назад и кто проходит сейчас уже сайт будет собираться и выглядеть на визуально по другому однозначно, и я уверена нервики вы будете славливать и не понимать почему не так как на видео получается хотя все четко повторяете , но увы это уже всё такие вопросы не к данному преподавателю, а конкретно к самой вебфлоу (команде разработчиков).

Я очень надеюсь что вы перезапишите этот курс заново или хотя бы вставками из за устаревости интерфейса, но понимаю что это накладно может быть.



### Вопрос по структуре проекта

У меня возник вопрос, связанный с созданием новых секций. Кроме первой секции Hero Section, на данный момент есть Section adventure и Section Video, а также техническая Design-system (об удалении опустившей секции Components тоже не говорилось). Причём каждая последующая секция после Hero Section опиралась на растягиваемый фрейм первой, из-за чего впоследствии приходилось вручную потом вынимать все новые элементы, чтобы они были организованы в отдельный, не вложенный в первый, фрейм. Действительно ли это так? Я вроде не мог просмотреть момент пристыковки снизу предыдущего фрейма пустого для новой секции, создание каждой новой начинается с растягивания старого фрейма и копирования на него элементов. Кроме того, обратил внимание, что в начале урока в списке фреймов появился Travel - более высокого уровня, чем все остальные. О нём ничего не говорилось.

<https://www.figma.com/design/cUjfBEDZAhiS7OSOF145iZ/Travel?node-id=9-2&t=PWkJWsnhsZbzMUwj-1>

1 ОТВЕТ

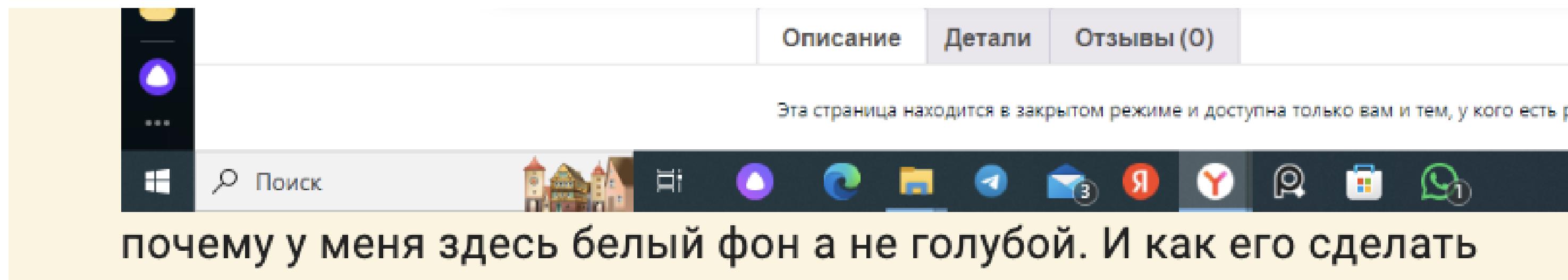


Dmitrii Fokeev Через несколько секунд

Привет! Перефразируйте вопрос на 1-2 строчки пожалуйста. В чём у вас проблема в структуре? Что то не работает?

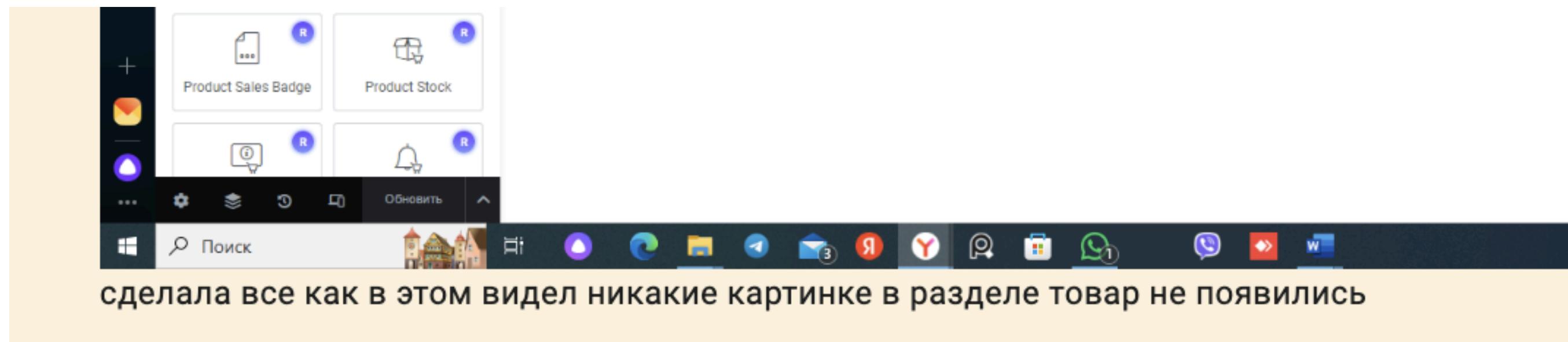
0

Не задавайте вопросы типа “не работает, что делать?”



Я смогу помочь только если вы пришлёте подробный и чёткий вопрос, а также сопутствующие материалы: скриншоты, видеозапись вашего экрана или ссылку на ваш проект с кодом.

Не задавайте вопросы типа  
“Сделал как в видео но не работает”

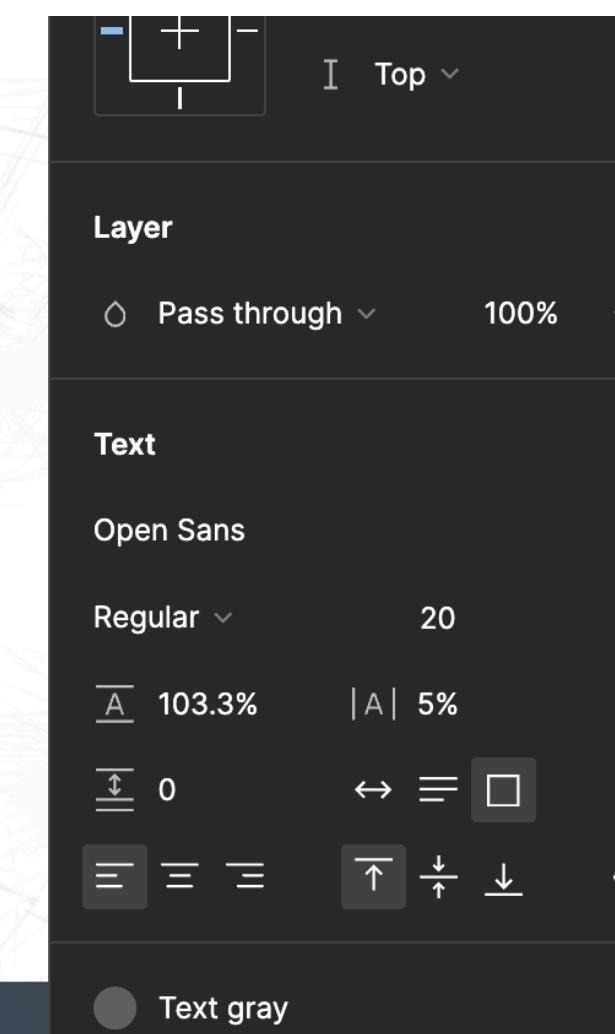
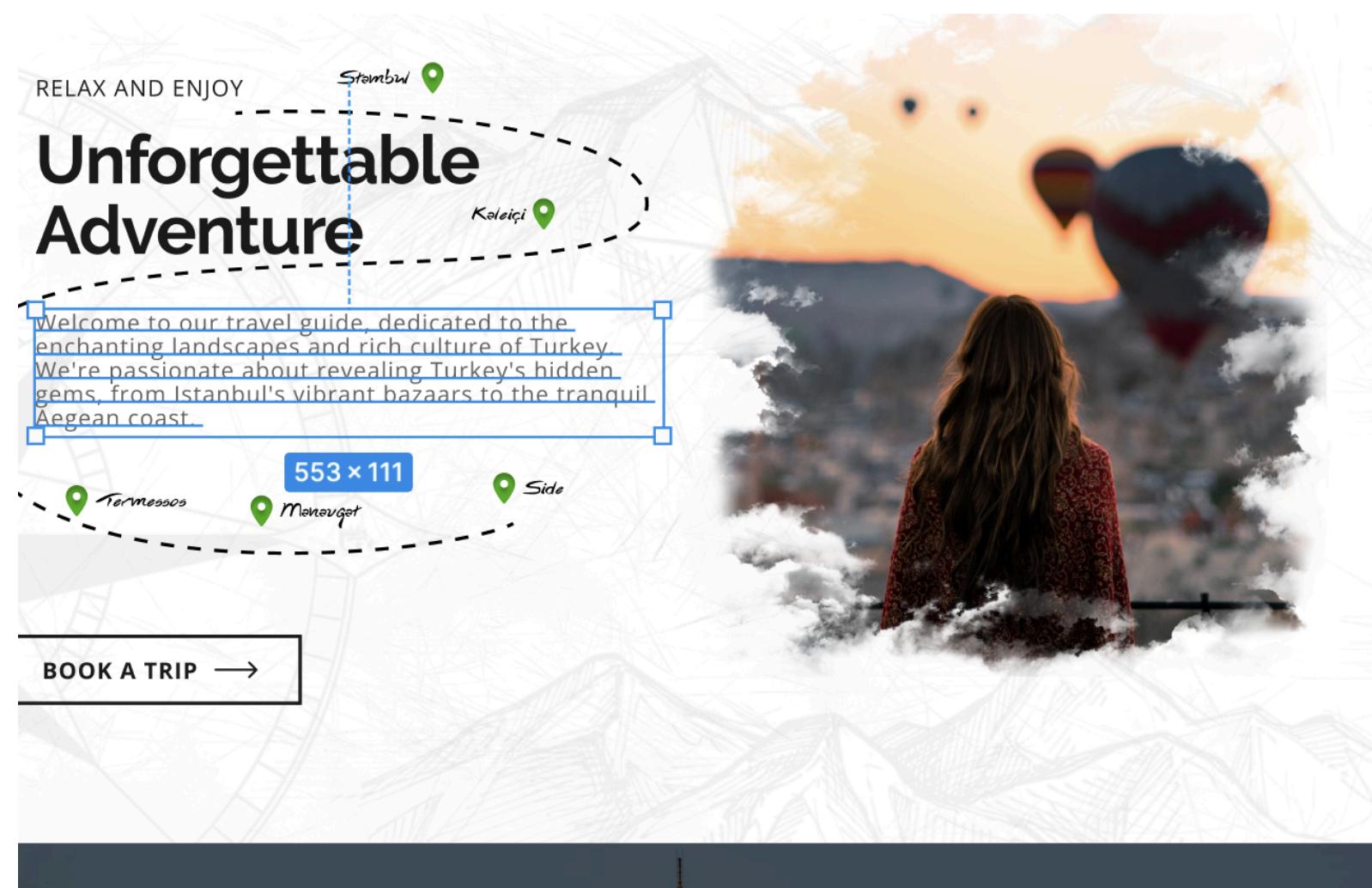


1. Ознакомьтесь с комментариями под видео.
2. Сделайте вторую и третью попытку, повторяя всё медленно и точно так, как показано в видео.  
В 99,9% случаев причиной является опечатка или ошибка.

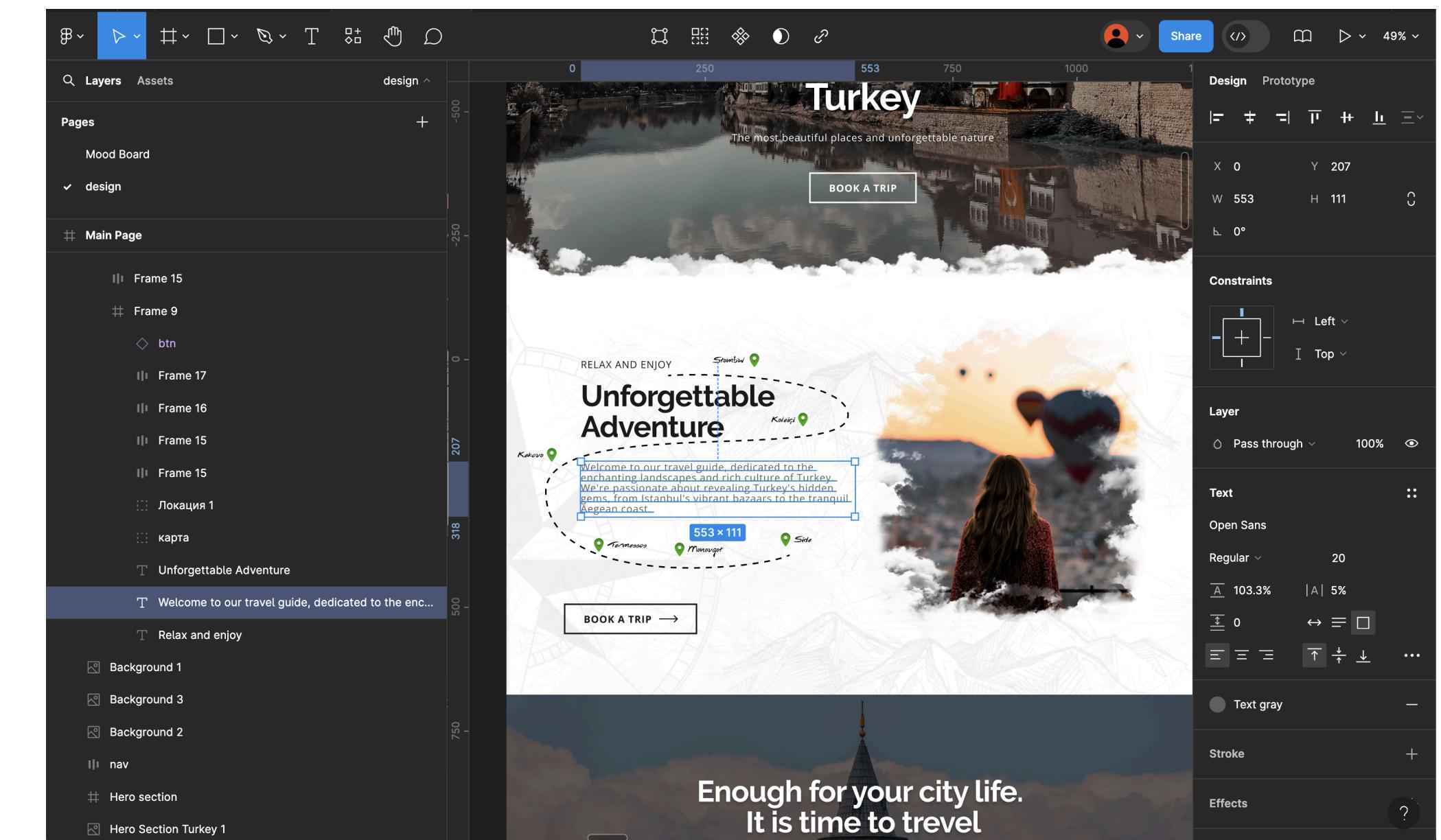
Если что-то работает в моём видео, оно должно работать и у вас.

Не присыпайте обрезанные скриншоты. (Я приближу нужное место)

Не так



Так



## Сравните ваш код с моим

В материалах курса всегда доступен весь код, который мы разбираем на уроках, а также готовый код из заданий. В случае возникновения ошибок сравните ваш код с предоставленным.

# Как проходить курс

– 3 варианта

## Видео курс

— Весь курс состоит из коротких видеоуроков в формате «смотри — повторяй»

## Курс **живой**

Если вы просматриваете курс на одной из официальных площадок — это значит, что автор модерирует курс ежедневно: все комментарии и пожелания учитываются и исправляются, тем самым огромное комьюнити студентов влияет на качество курса ежедневно.

## Курс с вами **навсегда**

Доступ к курсу не ограничен по времени.

## Курс **обновляется**

Если материал устаревает, автор его обновляет по мере возможности. Курс всегда свежий.

## **Нет домашек - есть практика**

В курсе множество практических заданий на отработку материала. Выполнять их или нет, решать вам. Вы уже взрослые. Тоже касается тестов. Но если есть желание поделиться результатом — это всегда приветствуется.

## Методичка

— Для курса создана специальная интерактивная методичка со всей теорией, памятками и правилами.

### Быстрый поиск

Используйте ctrl / cmd + f

### Копируйте код из комментариев

В комментариях к некоторым темам есть код, который вы можете скопировать и поэкспериментировать с ним.

### Версия для печати в материалах к курсу

В комментариях к некоторым темам есть код, который вы можете скопировать и поэкспериментировать с ним.

## **ЛЕГКИЙ И БЫСТРЫЙ РЕЗУЛЬТАТ**

2-3 месяца на обучение

### **Обзор возможностей React**

— Обзор функций и возможностей React на простых примерах.

Я показываю и рассказываю - вы повторяете код.  
(Желательно несколько раз и спустя время)

## **СРЕДНЯЯ СЛОЖНОСТЬ БОЛЬШЕ ВРЕМЕНИ**

3-4 месяца на обучение

### **Доп. практика и тесты**

— Всё из предыдущего пункта + практические задания и тесты на отработку пройденного материала.

## **СЛОЖНО, МЕСТАМИ НУДНО И ДОЛГО**

4-5 месяцев на обучение

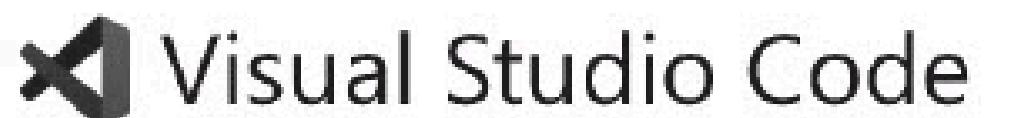
### **Создание проектов**

— Всё из прошлых двух пунктов + создание проектов.

# VS Code

– Горячие клавиши

# Горячие клавиши Mac:



Keyboard shortcuts for macOS

## General

⌘P, F1	Show Command Palette
⌘P	Quick Open, Go to File...
⌘N	New window-instance
⌘W	Close window-instance
⌘,	User Settings
⌘K ⌘S	Keyboard Shortcuts

## Basic editing

⌘X	Cut line (empty selection)
⌘C	Copy line (empty selection)
⌃↓ / ⌄↑	Move line down/up
⌃↖ / ⌄↖	Copy line down/up
⌘K	Delete line
⌘Enter / ⌄⌘Enter	Insert line below/above
⌘L	Jump to matching bracket
⌘] / ⌄[	Indent/outdent line
Home / End	Go to beginning/end of line
⌘↑ / ⌄↓	Go to beginning/end of file
⌃PgUp / ⌄PgUp	Scroll line up/down
⌃PgUp / ⌄PgDn	Scroll page up/down
⌃⌘[ / ⌄⌘]	Fold/unfold region
⌘K ⌄⌘[ / ⌄⌘]	Fold/unfold all subregions
⌘K ⌄⌘[ / ⌄⌘J	Fold/unfold all regions
⌘K ⌄⌘C	Add line comment
⌘K ⌄⌘U	Remove line comment
⌃/	Toggle line comment
⌃⌘A	Toggle block comment
⌃Z	Toggle word wrap

## Multi-cursor and selection

⌃ + click	Insert cursor
⌃⌘↑	Insert cursor above
⌃⌘↓	Insert cursor below
⌘U	Undo last cursor operation
⌃↖	Insert cursor at end of each line selected
⌘L	Select current line
⌘EL	Select all occurrences of current selection
⌘F2	Select all occurrences of current word
⌃⌘→ / ⌄←	Expand / shrink selection
⌃+ drag mouse	Column (box) selection
⌃⌘↑ / ⌄↓	Column (box) selection up/down
⌃⌘← / ⌄→	Column (box) selection left/right
⌃⌘PgUp	Column (box) selection page up
⌃⌘PgDn	Column (box) selection page down

## Search and replace

⌘F	Find
⌃⌘F	Replace
⌘G / ⌄⌘G	Find next/previous
⌃Enter	Select all occurrences of Find match
⌘D	Add selection to next Find match
⌘K ⌄D	Move last selection to next Find match

## Rich languages editing

⌃Space, ⌄I	Trigger suggestion
⌃Space	Trigger parameter hints
⌃F	Format document
⌘K ⌄F	Format selection
F12	Go to Definition
⌃F12	Peek Definition
⌘K F12	Open Definition to the side
⌘.	Quick Fix
⌃F12	Show References
F2	Rename Symbol
⌘K ⌄X	Trim trailing whitespace
⌘K M	Change file language

## Navigation

⌘T	Show all Symbols
⌃G	Go to Line...
⌘P	Go to File...
⌘O	Go to Symbol...
⌘M	Show Problems panel
F8 / ⌄F8	Go to next/previous error or warning
⌃Tab	Navigate editor group history
⌃- / ⌄-	Go back/forward
⌃M	Toggle Tab moves focus

## Editor management

⌘W	Close editor
⌘K F	Close folder
⌘I	Split editor
⌘I / ⌄⌘I / ⌄⌘I	Focus into 1st, 2nd, 3rd editor group
⌘K ⌄⌘→ / ⌄⌘→	Focus into previous/next editor group
⌘K ⌄⌘← / ⌄⌘←	Move editor left/right
⌘K ← / ⌄⌘→	Move active editor group

## File management

⌘N	New File
⌘O	Open File...
⌘S	Save
⌘S	Save As...
⌃⌘S	Save All
⌘W	Close
⌘K ⌄W	Close All

⌘T	Reopen closed editor
⌘K Enter	Keep preview mode editor open
⌃Tab / ⌄⌃Tab	Open next / previous
⌘K P	Copy path of active file
⌘K R	Reveal active file in Finder
⌘K O	Show active file in new window/instance

Display	
⌃⌘F	Toggle full screen
⌃⌘0	Toggle editor layout (horizontal/vertical)
⌘- / ⌄⌘-	Zoom in/out
⌘B	Toggle Sidebar visibility
⌘E	Show Explorer / Toggle focus
⌘F	Show Search
⌃⌘G	Show Source Control
⌘D	Show Debug
⌘X	Show Extensions
⌘H	Replace in files
⌘J	Toggle Search details
⌘U	Show Output panel
⌘V	Open Markdown preview
⌘K V	Open Markdown preview to the side
⌘Z	Zen Mode (Esc Esc to exit)

Debug	
F9	Toggle breakpoint
F5	Start/Continue
F11 / ⌄F11	Step into/ out
F10	Step over
⌃F5	Stop
⌘I	Show hover

Integrated terminal	
⌃`	Show integrated terminal
⌃⌃`	Create new terminal
⌘C	Copy selection
⌘↑ / ⌄↓	Scroll up/down
PgUp / PgDn	Scroll page up/down
⌘Home / End	Scroll to top/bottom

Other operating systems' keyboard shortcuts and additional unassigned shortcuts available at [aka.ms/vscodekeybindings](https://aka.ms/vscodekeybindings)

# Горячие клавиши Windows:



Keyboard shortcuts for Windows

## General

<b>Ctrl+Shift+P, F1</b>	Show Command Palette
<b>Ctrl+P</b>	Quick Open, Go to File...
<b>Ctrl+Shift+N</b>	New window-instance
<b>Ctrl+Shift+W</b>	Close window-instance
<b>Ctrl+,</b>	User Settings
<b>Ctrl+K Ctrl+S</b>	Keyboard Shortcuts

## Basic editing

<b>Ctrl+X</b>	Cut line (empty selection)
<b>Ctrl+C</b>	Copy line (empty selection)
<b>Alt+↑ / ↓</b>	Move line up/down
<b>Shift+Alt+↑ / ↓</b>	Copy line up/down
<b>Ctrl+Shift+K</b>	Delete line
<b>Ctrl+Enter</b>	Insert line below
<b>Ctrl+Shift+Enter</b>	Insert line above
<b>Ctrl+Shift+\`</b>	Jump to matching bracket
<b>Ctrl+] / [</b>	Indent/outdent line
<b>Home / End</b>	Go to beginning/end of line
<b>Ctrl+Home</b>	Go to beginning of file
<b>Ctrl+End</b>	Go to end of file
<b>Ctrl+↑ / ↓</b>	Scroll line up/down
<b>Alt+PgUp / PgDn</b>	Scroll page up/down
<b>Ctrl+Shift+[</b>	Fold (collapse) region
<b>Ctrl+Shift+]</b>	Unfold (uncollapse) region
<b>Ctrl+K Ctrl+[</b>	Fold (collapse) all subregions
<b>Ctrl+K Ctrl+]</b>	Unfold (uncollapse) all subregions
<b>Ctrl+K Ctrl+0</b>	Fold (collapse) all regions
<b>Ctrl+K Ctrl+J</b>	Unfold (uncollapse) all regions
<b>Ctrl+K Ctrl+C</b>	Add line comment
<b>Ctrl+K Ctrl+U</b>	Remove line comment
<b>Ctrl+/</b>	Toggle line comment
<b>Shift+Alt+A</b>	Toggle block comment
<b>Alt+Z</b>	Toggle word wrap

## Navigation

<b>Ctrl+T</b>	Show all Symbols
<b>Ctrl+G</b>	Go to Line...
<b>Ctrl+P</b>	Go to File...
<b>Ctrl+Shift+O</b>	Go to Symbol...
<b>Ctrl+Shift+M</b>	Show Problems panel
<b>F8</b>	Go to next error or warning
<b>Shift+F8</b>	Go to previous error or warning
<b>Ctrl+Shift+Tab</b>	Navigate editor group history
<b>Alt+← / →</b>	Go back / forward

**Ctrl+M**

Toggle Tab moves focus

## Search and replace

<b>Ctrl+F</b>	Find
<b>Ctrl+H</b>	Replace
<b>F3 / Shift+F3</b>	Find next/previous
<b>Alt+Enter</b>	Select all occurrences of Find match
<b>Ctrl+D</b>	Add selection to next Find match
<b>Ctrl+K Ctrl+D</b>	Move last selection to next Find match
<b>Alt+C / R / W</b>	Toggle case-sensitive / regex / whole word

## Multi-cursor and selection

<b>Alt+Click</b>	Insert cursor
<b>Ctrl+Alt+↑ / ↓</b>	Insert cursor above / below
<b>Ctrl+U</b>	Undo last cursor operation
<b>Shift+Alt+I</b>	Insert cursor at end of each line selected
<b>Ctrl+L</b>	Select current line
<b>Ctrl+Shift+L</b>	Select all occurrences of current selection
<b>Ctrl+F2</b>	Select all occurrences of current word
<b>Shift+Alt+→</b>	Expand selection
<b>Shift+Alt+←</b>	Shrink selection
<b>Shift+Alt+ (drag mouse)</b>	Column (box) selection
<b>Ctrl+Shift+Alt+ (arrow key)</b>	Column (box) selection
<b>Ctrl+Shift+Alt+PgUp/PgDn</b>	Column (box) selection page up/down

## Rich languages editing

<b>Ctrl+Space</b>	Trigger suggestion
<b>Ctrl+Shift+Space</b>	Trigger parameter hints
<b>Shift+Alt+F</b>	Format document
<b>Ctrl+K Ctrl+F</b>	Format selection
<b>F12</b>	Go to Definition
<b>Alt+F12</b>	Peek Definition
<b>Ctrl+K F12</b>	Open Definition to the side
<b>Ctrl+. .</b>	Quick Fix
<b>Shift+F12</b>	Show References
<b>F2</b>	Rename Symbol
<b>Ctrl+K Ctrl+X</b>	Trim trailing whitespace
<b>Ctrl+K M</b>	Change file language

## Editor management

<b>Ctrl+F4, Ctrl+W</b>	Close editor
<b>Ctrl+K F</b>	Close folder
<b>Ctrl+\`</b>	Split editor
<b>Ctrl+1 / 2 / 3</b>	Focus into 1 <sup>st</sup> , 2 <sup>nd</sup> or 3 <sup>rd</sup> editor group
<b>Ctrl+K Ctrl+←/→</b>	Focus into previous/next editor group
<b>Ctrl+Shift+PgUp / PgDn</b>	Move editor left/right
<b>Ctrl+K ← / →</b>	Move active editor group

## File management

<b>Ctrl+N</b>	New File
<b>Ctrl+O</b>	Open File...
<b>Ctrl+S</b>	Save
<b>Ctrl+Shift+S</b>	Save As...
<b>Ctrl+K S</b>	Save All
<b>Ctrl+F4</b>	Close
<b>Ctrl+K Ctrl+W</b>	Close All
<b>Ctrl+Shift+T</b>	Reopen closed editor
<b>Ctrl+K Enter</b>	Keep preview mode editor open
<b>Ctrl+Tab</b>	Open next
<b>Ctrl+Shift+Tab</b>	Open previous
<b>Ctrl+K P</b>	Copy path of active file
<b>Ctrl+K R</b>	Reveal active file in Explorer
<b>Ctrl+K O</b>	Show active file in new window-instance

## Display

<b>F11</b>	Toggle full screen
<b>Shift+Alt+0</b>	Toggle editor layout (horizontal/vertical)
<b>Ctrl+= -</b>	Zoom in/out
<b>Ctrl+B</b>	Toggle Sidebar visibility
<b>Ctrl+Shift+E</b>	Show Explorer / Toggle focus
<b>Ctrl+Shift+F</b>	Show Search
<b>Ctrl+Shift+G</b>	Show Source Control
<b>Ctrl+Shift+D</b>	Show Debug
<b>Ctrl+Shift+X</b>	Show Extensions
<b>Ctrl+Shift+H</b>	Replace in files
<b>Ctrl+Shift+J</b>	Toggle Search details
<b>Ctrl+Shift+U</b>	Show Output panel
<b>Ctrl+Shift+V</b>	Open Markdown preview
<b>Ctrl+K V</b>	Open Markdown preview to the side
<b>Ctrl+K Z</b>	Zen Mode (Esc Esc to exit)

## Debug

<b>F9</b>	Toggle breakpoint
<b>F5</b>	Start/Continue
<b>Shift+F5</b>	Stop
<b>F11 / Shift+F11</b>	Step into/out
<b>F10</b>	Step over
<b>Ctrl+K Ctrl+I</b>	Show hover

## Integrated terminal

<b>Ctrl+`</b>	Show integrated terminal
<b>Ctrl+Shift+`</b>	Create new terminal
<b>Ctrl+C</b>	Copy selection
<b>Ctrl+V</b>	Paste into active terminal
<b>Ctrl+↑ / ↓</b>	Scroll up/down
<b>Shift+PgUp / PgDn</b>	Scroll page up/down
<b>Ctrl+Home / End</b>	Scroll to top/bottom

Other operating systems' keyboard shortcuts and additional unassigned shortcuts available at [aka.ms/vscodekeybindings](http://aka.ms/vscodekeybindings)

# Основы терминала

– Базовые команды и горячие клавиши

## Если у вас Mac OS

То у вас один терминал независимо от версии ОС.

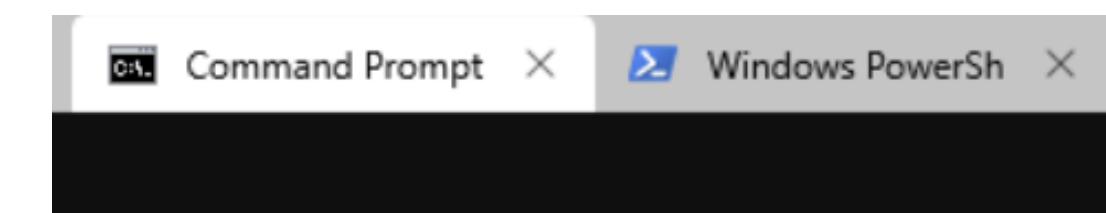
Все OK, разбираться ни с чем не нужно.

## Если у вас Windows

То у вас установлен один из 3-х вариантов программы терминала:

1. Command Prompt (cmd.exe)
2. PowerShell
3. Windows Terminal

Узнать, какой у вас терминал, можно по названию окна программы терминала после запуска (терминал в VS Code также подписан).



# macOS Terminal (Terminal.app) и Linux Terminal:

## Папки и пути:

1. Просмотр содержимого директории:

- **ls**: Показывает список файлов и папок.

2. Смена директории:

- **cd /Users/Username/Documents**: Перемещает в указанную директорию.

3. Копирование файлов:

- **cp file.txt /Volumes/Backup/file.txt**: Копирует файл в указанное место.

4. Удаление файлов:

- **rm file.txt**: Удаляет файл.

5. Перемещение файлов:

- **mv file.txt /Volumes/Backup/**: Перемещает файл.

6. Создание папки:

- **mkdir NewFolder**: Создает новую папку.

7. Удаление папки:

- **rmdir NewFolder**: Удаляет пустую папку.

8. Создание файла:

- **touch newfile.txt**: Создает пустой файл.

9. Очистка экрана:

- **clear**: Очищает экран терминала.

10. Выход из терминала:

- **exit**: Завершает сеанс терминала.

## Горячие клавиши:

1. Очистка экрана: **Ctrl + L**.

2. История команд: Стрелка вверх/вниз для просмотра предыдущих команд.

3. Прерывание команды: **Ctrl + C**.

4. Автозаполнение: **Tab**.

5. Поиск в истории команд: **Ctrl + R**.

6. Перемещение курсора по словам: **Alt + ←/→**.

7. Удаление слова перед курсором: **Ctrl + W**.

8. Удаление строки до курсора: **Ctrl + U**.

# Если у вас Command Prompt (cmd.exe) на Windows:

## Папки и пути:

1. Просмотр содержимого директории:

- `dir`: Показывает список файлов и папок в текущей директории.

2. Смена директории:

- `cd C:\Users\Username\Documents`: Перемещает в указанную директорию.

3. Копирование файлов:

- `copy file.txt D:\Backup\file.txt`: Копирует файл в указанное место.

4. Удаление файлов:

- `del file.txt`: Удаляет файл.

5. Перемещение файлов:

- `move file.txt D:\Backup\`: Перемещает файл.

6. Создание папки:

- `mkdir NewFolder`: Создает новую папку.

7. Удаление папки:

- `rmdir NewFolder`: Удаляет пустую папку.

8. Создание файла:

- `type nul > newfile.txt`: Создает пустой файл.

9. Очистка экрана:

- `cls`: Очищает экран терминала.

10. Выход из терминала:

- `exit`: Завершает сеанс командной строки.

## Горячие клавиши:

1. Очистка экрана: `Ctrl + L` (иногда не работает, тогда используйте команду `cls`).

2. История команд: Стрелка вверх/вниз для просмотра предыдущих команд.

3. Прерывание команды: `Ctrl + C`.

4. Автозаполнение: `Tab`.

# Если у вас PowerShell на Windows:

## Папки и пути:

1. Просмотр содержимого директории:

- `Get-ChildItem`: Показывает список файлов и папок.

2. Смена директории:

- `Set-Location C:\Users\Username\Documents`: Перемещает в указанную директорию.

3. Копирование файлов:

- `Copy-Item file.txt D:\Backup\file.txt`: Копирует файл в указанное место.

4. Удаление файлов:

- `Remove-Item file.txt`: Удаляет файл.

5. Перемещение файлов:

- `Move-Item file.txt D:\Backup\`: Перемещает файл.

6. Создание папки:

- `New-Item -ItemType Directory -Name NewFolder`: Создает новую папку.

7. Удаление папки:

- `Remove-Item NewFolder`: Удаляет папку.

8. Создание файла:

- `New-Item -ItemType File -Name newfile.txt`: Создает пустой файл.

9. Очистка экрана:

- `Clear-Host`: Очищает экран терминала.

10. Выход из PowerShell:

- `Exit-PSSession`: Завершает сеанс PowerShell.

## Горячие клавиши:

1. Очистка экрана: **Ctrl + L**

2. История команд: **Стрелка вверх/вниз** для просмотра предыдущих команд.

3. Прерывание команды: **Ctrl + C**.

4. Автозаполнение: **Tab**.

## TERMINAL COMMANDS

Windows Terminal Поддерживает все команды, перечисленные выше для cmd.exe и PowerShell.

# Проблема с правами доступа на macOS

— используй sudo

# Ошибка при установке

— иногда, на компьютерах Mac, при установке пакетов, или использования команд в терминале, вместо результата вы увидите ошибку.

## Например:

установка нового проекта через **npm create vite**

Как правило - это ошибка доступа. У вас просто нет прав администратора в текущей учетной записи

```

Creating a new React app in /Users/dmitrijfokeev/Desktop/pop1.
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...
npm error code EEXIST
npm error syscall rename
npm error path /Users/dmitrijfokeev/.npm/_cacache/tmp/026ceb71
npm error dest /Users/dmitrijfokeev/.npm/_cacache/content-v2/sha512/8c/a8/6a864be1a96b24ee1fba24b18579b130cbda444997f16
ab53e7d076f0b9cd0d9d560fab2d141349fd59284c98ce721e61fcda5ab0f7f7901b39cc112b59
npm error errno EEXIST
npm error Invalid response body while trying to fetch https://registry.npmjs.org/react-refresh: EACCES: permission denied, rename '/Users/dmitrijfokeev/.npm/_cacache/tmp/026ceb71' -> '/Users/dmitrijfokeev/.npm/_cacache/content-v2/sha512/8
c/a8/6a864be1a96b24ee1fba24b18579b130cbda444997f16a53e7d076f0b9cd0d9d560fab2d141349fd59284c98ce721e61fcda5ab0f7f7901b
39cc112b59'
npm error File exists: /Users/dmitrijfokeev/.npm/_cacache/content-v2/sha512/8c/a8/6a864be1a96b24ee1fba24b18579b130cbda4
44997f16a53e7d076f0b9cd0d9d560fab2d141349fd59284c98ce721e61fcda5ab0f7f7901b39cc112b59
npm error Remove the existing file and try again, or run npm
npm error with --force to overwrite files recklessly.
npm error A complete log of this run can be found in: /Users/dmitrijfokeev/.npm/_logs/2024-08-29T10_17_35_537Z-debug-0.log
Aborting installation.
  npm install --no-audit --save --save-exact --loglevel error react-dom react-scripts cra-template has failed.
Deleting generated file... package.json
Deleting pop1/ from /Users/dmitrijfokeev/Desktop
Done.

```

## 2 варианта решения:

1. Перед командой следует писать ключевое слово **sudo**.

Тогда программа запросит пароль и выполнит установку от имени администратора. Однако впоследствии могут возникнуть сложности при удалении или изменении файлов в проекте. (При удалении файлов могут потребоваться дополнительные запросы на доступ администратора.)

## Пример:

`~/Desktop $sudo npm create vite`

2. Используйте в терминале команду, которая предоставит права администратора учетной записи, в которой вы находитесь (вы можете скопировать её отсюда или из комментариев к слайду):

`sudo chown -R $(whoami) $(npm config get prefix)/{lib/node_modules,bin,share}`

# Чит лист JS

– Основная памятка по JS

# Оператор switch

- вместо *if*
- это конструкция управления потоком в JavaScript, которая позволяет выполнить одну из нескольких возможных операций в зависимости от значения выражения.

Он особенно полезен, когда нужно сравнить одно значение с несколькими возможными вариантами и выполнить соответствующий код.

— Можете скопировать код из комментариев ниже и протестировать

## Оператор **switch**:

```
switch (expression) {  
  case value1:  
    // Код, который будет выполнен, если expression === value1  
    break;  
  case value2:  
    // Код, который будет выполнен, если expression === value2  
    break;  
  // ...  
  default:  
    // Код, который будет выполнен, если ни одно из значений не совпало  
}
```

### Как работает switch?

#### 1. Вычисление выражения:

- Выражение, указанное в скобках после `switch`, вычисляется один раз.
- Результат этого выражения затем последовательно сравнивается со значениями, указанными в `case`.

#### 2. Сравнение с case:

- `switch` последовательно сравнивает значение выражения с каждым значением, указанным в `case`.
- Как только найдено совпадение, выполняется соответствующий блок кода.

#### 3. Выполнение кода:

- Когда найдено совпадение, выполняется код внутри соответствующего `case`.
- Оператор `break` используется для прекращения выполнения последующих блоков `case`. Если `break` отсутствует, `switch` продолжит выполнение кода в следующем `case`, даже если совпадение уже найдено (это называется “провал через”).

#### 4. Блок default:

- Блок `default` выполняется, если ни одно из значений `case` не совпало с выражением.
- Этот блок необязателен, но он полезен для обработки всех случаев, которые не были предусмотрены.

# Методы массива:

## Методы для добавления и удаления элементов:

- push()**: Добавляет один или несколько элементов в конец массива и возвращает новую длину массива. `array.push(element1, ..., elementN)`
- pop()**: Удаляет последний элемент из массива и возвращает его. `array.pop()`
- unshift()**: Добавляет один или несколько элементов в начало массива и возвращает новую длину массива. `array.unshift(element1, ..., elementN)`
- shift()**: Удаляет первый элемент из массива и возвращает его. `array.shift()`

## Методы для перебора элементов:

- forEach()**: Выполняет указанную функцию один раз для каждого элемента массива. `array.forEach((item, index, arr) =>{})`
- map()**: Создает новый массив с результатами вызова указанной функции для каждого элемента массива. `array.map((item, index, arr) =>{})`
- filter()**: Создает новый массив со всеми элементами, прошедшими проверку, задаваемую в передаваемой функции. `array.filter((item, index, arr) =>{})`
- reduce()**: Применяет функцию к аккумулятору и каждому элементу массива (слева направо), сводя его к одному значению. `array.reduce((accumulator, item, index, arr) => {}, initialValue)`

## Методы для поиска элементов:

- find()**: Возвращает первое значение, которое проходит проверку в функции. `array.find((item, index, arr) =>{})`
- findIndex()**: Возвращает индекс первого элемента в массиве, который проходит проверку в функции. `array.findIndex((item, index, arr) =>{})`
- includes()**: Определяет, содержит ли массив определённый элемент, возвращая `true` или `false`. `array.includes(element)`
- indexOf()**: Возвращает первый индекс, по которому данный элемент может быть найден в массиве, или `-1`, если элемент не найден. `array.indexOf(element)`

## Методы для изменения элементов:

- slice()**: Возвращает новый массив, содержащий копию части исходного массива. `array.slice(begin, end)`
- splice()**: Изменяет содержимое массива, удаляя существующие элементы и/или добавляя новые. `array.splice(start, deleteCount, item1, ..., itemN)`
- concat()**: Возвращает новый массив, состоящий из массива, на котором он был вызван, соединенного с другими массивами и/или значениями. `array.concat(value1, ..., valueN)`
- join()**: Объединяет все элементы массива в строку. `array.join(separator)`

## Методы для проверки условий:

- every()**: Проверяет, удовлетворяют ли все элементы массива условию, заданному в функции. `array.every(item, index, arr =>{})`
- some()**: Проверяет, удовлетворяет ли какой-либо элемент массива условию, заданному в функции. `array.some(item, index, arr =>{})`

## Метод sort:

- Сортирует элементы массива по порядку алфавита. Порядок сортировки по умолчанию соответствует порядку кодовых точек Unicode

**1-Метод sort()** изменяет оригинальный массив.

Поэтому старайтесь работать с копиями массивов.

Для массивов, содержащих строки, работает просто по алфавиту  
(Заглавные буквы имеют больший приоритет)

**2-Метод sort()** конвертирует элементы в строку и символы цифр будут сортироваться не по числам, а по приоритетности строковых символов

Поэтому, для сортировки чисел нужно использовать call-back функцию с параметрами `a` и `b`

С помощью этих параметров сравниваются все числа массива.  
Если число `a` больше, чем число `b`, то поменяй местами элементы массива  
Если число `a` меньше, чем число `b`, то сохраняй порядок, как есть

# Отдельно про метод sort:

Метод sort:

- Сортирует элементы массива по порядку алфавита. Порядок сортировки по умолчанию соответствует порядку кодовых точек Unicode

1-Метод sort() изменяет оригинальный массив.

Поэтому старайтесь работать с копиями массивов.

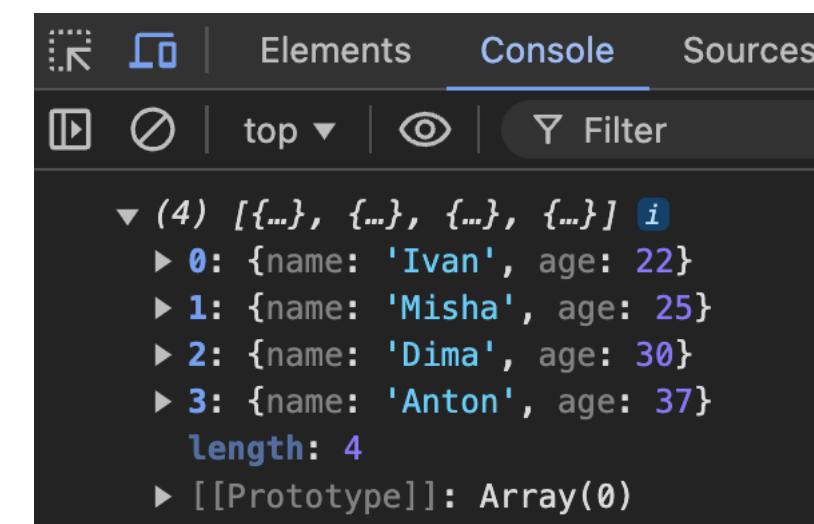
Для массивов, содержащих только строки, работает просто по алфавиту (Заглавные буквы имеют больший приоритет)

```
const arr = ["d", "e", "b", "t", "r", "x", "s", "a", "g", "u"];
sortedArr = [...arr].sort();
console.log(sortedArr); // ['a', 'b', 'd', 'e', 'g', 'r', 's', 't', 'u', 'x']
```

2- a и b – это два элемента массива, которые сравниваются в процессе сортировки. Каждый элемент массива является объектом, содержащим, свойство age.

- Функция сравнения (a, b) => a.age - b.age определяет порядок сортировки элементов в массиве:
- Если результат вычисления меньше нуля, то элемент a должен быть расположен перед элементом b.
- Если результат равен нулю, порядок элементов a и b не изменяется
- Если результат больше нуля, элемент b должен быть расположен перед элементом a.

```
const arr = [
  {name: "Dima", age: 30},
  {name: "Ivan", age: 22},
  {name: "Misha", age: 25},
  {name: "Anton", age: 37},
];
//Сортируем по возрасту:
const sortedArr = [...arr].sort((a, b) => a.age - b.age);
console.log(sortedArr);
```



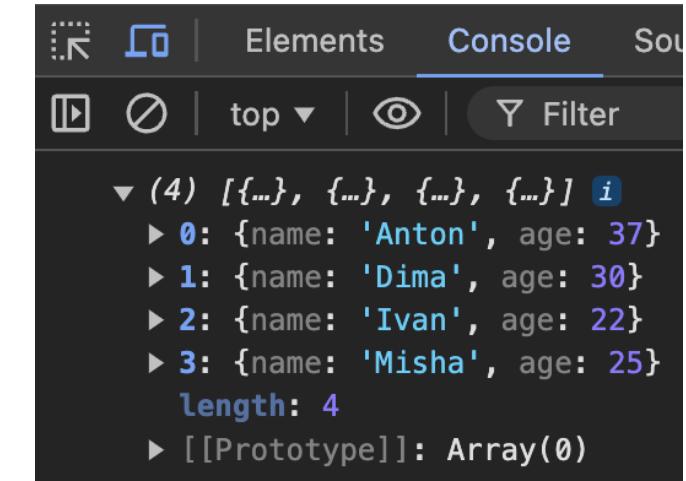
Создаем независимую копию массива так как метод sort изменяет оригинальный массив. А это не рекомендуется

3. Сортировка по алфавиту в объекте

Для сравнения нужен метод localeCompare()

Метод localeCompare() – это метод строкового объекта в JavaScript, который используется для сравнения двух строк в соответствии с локальными правилами сортировки

```
const arr = [
  {name: "Dima", age: 30},
  {name: "Ivan", age: 22},
  {name: "Misha", age: 25},
  {name: "Anton", age: 37},
];
//Сортируем по алфавиту:
const sortedArr = [...arr].sort((a, b) => a.name.localeCompare(b.name));
console.log(sortedArr);
```



localeCompare() сравнивает строку на которой он вызывается с переданной строкой (аргументом метода) и возвращает числовое значение:

- Если вызывающая строка должна располагаться перед переданной строкой в порядке сортировки, метод возвращает отрицательное число.
  - Если строки идентичны с точки зрения локальной сортировки, метод возвращает 0.
  - Если вызывающая строка должна идти после переданной строки, метод возвращает положительное число.
2. Метод localeCompare() учитывает локаль и языковые особенности при сравнении, включая порядок букв, чувствительность к регистру и акцентам.

## ОБЪЕКТ

# Методы объекта:

Методы для получения информации об объекте:

- **Object.keys()**: Возвращает массив, содержащий имена всех перечисляемых свойств объекта. Object.keys(obj)
- **Object.values()**: Возвращает массив, содержащий значения всех перечисляемых свойств объекта. Object.values(obj)
- **Object.entries()**: Возвращает массив, содержащий массивы с именами и значениями всех перечисляемых свойств объекта. Object.entries(obj)

Методы для работы с прототипами объектов:

- **Object.create()**: Создает новый объект с указанным прототипом и свойствами. Object.create(proto, [propertiesObject])
- **Object.getPrototypeOf()**: Возвращает прототип указанного объекта. Object.getPrototypeOf(obj)
- **Object.setPrototypeOf()**: Устанавливает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта. Object.setPrototypeOf(obj, prototype)

Методы для определения свойств объектов:

- **Object.defineProperty()**: Определяет новое или изменяет существующее свойство непосредственно на объекте. Object.defineProperty(obj, prop, descriptor)
- **Object.defineProperties()**: Определяет новые или изменяет существующие свойства непосредственно на объекте. Object.defineProperties(obj, props)
- **Object.getOwnPropertyDescriptor()**: Возвращает дескриптор свойства для указанного свойства объекта. Object.getOwnPropertyDescriptor(obj, prop)
- **Object.getOwnPropertyDescriptors()**: Возвращает все собственные дескрипторы свойств объекта. Object.getOwnPropertyDescriptors(obj)

Методы для работы с свойствами объектов:

- **Object.hasOwnProperty()**: Проверяет, является ли указанное свойство собственным свойством объекта (не наследуется). Object.hasOwnProperty(obj, prop)
- **Object.propertyIsEnumerable()**: Возвращает true, если указанное свойство объекта может перечисляться с помощью цикла for...in.  
obj.propertyIsEnumerable(prop)
- **Object.is()**: Определяет, являются ли два значения одним и тем же значением. Object.is(value1, value2)

Методы для создания и клонирования объектов:

- **Object.assign()**: Копирует все перечисляемые свойства одного или нескольких исходных объектов в целевой объект. Object.assign(target, ...sources)
- **Object.freeze()**: Замораживает объект: другой код не может удалить или изменить любые свойства. Object.freeze(obj)
- **Object.seal()**: Запечатывает объект, предотвращая добавление новых свойств к объекту и делая все существующие свойства не настраиваемыми.  
Object.seal(obj)

Методы для работы с метаданными объекта:

- **Object.isExtensible()**: Определяет, является ли объект расширяемым (то есть, можно ли к нему добавлять новые свойства). Object.isExtensible(obj)
- **Object.isFrozen()**: Определяет, был ли объект заморожен. Object.isFrozen(obj)
- **Object.isSealed()**: Определяет, является ли объект запечатанным. Object.isSealed(obj)
- **Object.preventExtensions()**: Предотвращает добавление новых свойств к объекту. Object.preventExtensions(obj)

# Методы строк:

Методы для поиска и извлечения подстрок:

- **charAt()**: Возвращает символ, стоящий на указанной позиции в строке. str.charAt(index)
- **charCodeAt()**: Возвращает числовое значение Юникода для символа, стоящего на указанной позиции в строке. str.charCodeAt(index)
- **concat()**: Соединяет указанные строки в одну строку. str.concat(string2, ..., stringN)
- **includes()**: Определяет, содержится ли одна строка внутри другой, возвращая true или false. str.includes(searchString, position)
- **indexOf()**: Возвращает индекс первого вхождения указанного значения в строке, или -1, если значение не найдено. str.indexOf(searchValue, fromIndex)
- **lastIndexOf()**: Возвращает индекс последнего вхождения указанного значения в строке, или -1, если значение не найдено. str.lastIndexOf(searchValue, fromIndex)
- **localeCompare()**: Сравнивает две строки в текущей локали и возвращает число, указывающее, должна ли строка сортироваться перед, после или быть на том же уровне, что и указанная строка. str.localeCompare(compareString)
- **match()**: Извлекает совпадения строки с регулярным выражением. str.match(regexp)
- **matchAll()**: Возвращает итератор всех совпадений строки с регулярным выражением, включая захваченные группы. str.matchAll(regexp)
- **search()**: Выполняет поиск совпадения между регулярным выражением и строкой, возвращая индекс первого совпадения, или -1, если совпадение не найдено. str.search(regexp)
- **slice()**: Извлекает часть строки и возвращает новую строку без изменения оригинала. str.slice(beginIndex, endIndex)
- **split()**: Разделяет строку на массив подстрок, используя указанный разделитель. str.split(separator, limit)
- **substring()**: Возвращает подстроку между двумя индексами или от указанного индекса до конца строки. str.substring(indexStart, indexEnd)
- **substr()**: Возвращает подстроку, начиная с указанного индекса и длиной, указанной вторым параметром. str.substr(start, length)

Методы для изменения строки:

- **toLowerCase()**: Преобразует строку в нижний регистр. str.toLowerCase()
- **toUpperCase()**: Преобразует строку в верхний регистр. str.toUpperCase()
- **trim()**: Удаляет пробельные символы с начала и конца строки. str.trim()
- **trimStart() / trimLeft()**: Удаляет пробельные символы с начала строки. str.trimStart()
- **trimEnd() / trimRight()**: Удаляет пробельные символы с конца строки. str.trimEnd()
- **padStart()**: Дополняет текущую строку другой строкой, до заданной длины, с начала текущей строки. str.padStart(targetLength, padString)
- **padEnd()**: Дополняет текущую строку другой строкой, до заданной длины, с конца текущей строки. str.padEnd(targetLength, padString)
- **repeat()**: Возвращает новую строку, содержащую указанное количество соединений строки, на которой был вызван метод. str.repeat(count)
- **replace()**: Заменяет подстроку или совпадения с регулярным выражением в строке на указанное значение. str.replace(regexp|substr, newSubstr|function)
- **replaceAll()**: Заменяет все совпадения с регулярным выражением или строкой на указанное значение. str.replaceAll(regexp|substr, newSubstr|function)

Методы для работы с Юникодом и кодовыми точками:

- **codePointAt()**: Возвращает неотрицательное целое значение кодовой точки Юникода на указанной позиции. str.codePointAt(pos)
- **normalize()**: Возвращает нормализованную форму строки в соответствии со спецификацией Юникода. str.normalize([form])

Методы для работы с шаблонами:

- **startsWith()**: Определяет, начинается ли строка с символов указанной строки, возвращая true или false. str.startsWith(searchString, position)
- **endsWith()**: Определяет, заканчивается ли строка символами указанной строки, возвращая true или false. str.endsWith(searchString, length)

# Методы чисел:

Методы объекта Number:

- **Number.isFinite()**: Определяет, является ли переданное значение конечным числом. `Number.isFinite(value)`
- **Number.isInteger()**: Определяет, является ли переданное значение целым числом. `Number.isInteger(value)`
- **Number.isNaN()**: Определяет, является ли переданное значение NaN (Not-a-Number). `Number.isNaN(value)`
- **Number.isSafeInteger()**: Определяет, является ли переданное значение безопасным целым числом (в пределах диапазона безопасных целых чисел). `Number.isSafeInteger(value)`
- **Number.parseFloat()**: Преобразует строку в число с плавающей запятой. `Number.parseFloat(string)`
- **Number.parseInt()**: Преобразует строку в целое число, используя указанную систему счисления. `Number.parseInt(string, radix)`

Методы прототипа Number:

- **toExponential()**: Возвращает строку с числом в экспоненциальной форме. `num.toExponential(fractionDigits)`
- **toFixed()**: Возвращает строку с числом, округленным до указанного числа знаков после запятой. `num.toFixed(digits)`
- **toLocaleString()**: Возвращает строку с числом в соответствии с языковыми настройками. `num.toLocaleString([locales[, options]])`
- **toPrecision()**: Возвращает строку с числом с указанной точностью. `num.toPrecision(precision)`
- **toString()**: Возвращает строку, представляющую объект Number. `num.toString([radix])`
- **valueOf()**: Возвращает примитивное значение объекта Number. `num.valueOf()`

Глобальные методы для работы с числами:

- **isNaN()**: Определяет, является ли переданное значение NaN (Not-a-Number). `isNaN(value)`
- **isFinite()**: Определяет, является ли переданное значение конечным числом. `isFinite(value)`
- **parseInt()**: Преобразует строку в целое число, используя указанную систему счисления. `parseInt(string, radix)`
- **parseFloat()**: Преобразует строку в число с плавающей запятой. `parseFloat(string)`

Свойства объекта Number:

- **Number.EPSILON**: Возвращает наименьшее значение, которое можно добавить к 1 и получить другое число. `Number.EPSILON`
- **Number.MAX\_SAFE\_INTEGER**: Возвращает наибольшее целое число, которое может быть точно представлено в JavaScript. `Number.MAX_SAFE_INTEGER`
- **Number.MAX\_VALUE**: Возвращает наибольшее число, которое может быть представлено в JavaScript. `Number.MAX_VALUE`
- **Number.MIN\_SAFE\_INTEGER**: Возвращает наименьшее целое число, которое может быть точно представлено в JavaScript. `Number.MIN_SAFE_INTEGER`
- **Number.MIN\_VALUE**: Возвращает наименьшее положительное число, которое может быть представлено в JavaScript. `Number.MIN_VALUE`
- **Number.NaN**: Представляет значение Not-a-Number (NaN). `Number.NaN`
- **Number.NEGATIVE\_INFINITY**: Представляет отрицательную бесконечность. `Number.NEGATIVE_INFINITY`
- **Number.POSITIVE\_INFINITY**: Представляет положительную бесконечность. `Number.POSITIVE_INFINITY`

# Методы Даты:

## Создание и получение текущей даты

- **new Date():** Создает объект даты, представляющий текущее время и дату. new Date()
- **Date.now():** Возвращает количество миллисекунд, прошедших с 1 января 1970 года (начало эпохи Unix). Date.now()

## Создание объекта даты с определенной датой и временем

- **new Date(milliseconds):** Создает объект даты, представляющий указанное количество миллисекунд, прошедших с 1 января 1970 года. new Date(1627873200000)
- **new Date(dateString):** Создает объект даты из строки, представляющей дату и время. new Date('2023-07-23')
- **new Date(year, month, day, hours, minutes, seconds, milliseconds):** Создает объект даты с указанными значениями. new Date(2023, 6, 23, 10, 30, 0, 0)

## Методы получения компонентов даты

- **getFullYear():** Возвращает год в виде четырехзначного числа. date.getFullYear()
- **getMonth():** Возвращает месяц от 0 до 11 (где 0 - январь, а 11 - декабрь). date.getMonth()
- **getDate():** Возвращает день месяца от 1 до 31. date.getDate()
- **getDay():** Возвращает день недели от 0 до 6 (где 0 - воскресенье, а 6 - суббота). date.getDay()
- **getHours():** Возвращает час от 0 до 23. date.getHours()
- **getMinutes():** Возвращает минуты от 0 до 59. date.getMinutes()
- **getSeconds():** Возвращает секунды от 0 до 59. date.getSeconds()
- **getMilliseconds():** Возвращает миллисекунды от 0 до 999. date.getMilliseconds()
- **getTime():** Возвращает количество миллисекунд, прошедших с 1 января 1970 года. date.getTime()
- **getTimezoneOffset():** Возвращает разницу в минутах между местным часовым поясом и UTC. date.getTimezoneOffset()

## Методы установки компонентов даты

- **setFullYear(year, [month], [day]):** Устанавливает год. Опционально можно указать месяц и день. date.setFullYear(2024, 6, 23)
- **setMonth(month, [day]):** Устанавливает месяц. Опционально можно указать день. date.setMonth(6, 23)
- **setDate(day):** Устанавливает день месяца. date.setDate(23)
- **setHours(hours, [minutes], [seconds], [milliseconds]):** Устанавливает часы, минуты, секунды и миллисекунды. date.setHours(10, 30, 0, 0)
- **setMinutes(minutes, [seconds], [milliseconds]):** Устанавливает минуты, секунды и миллисекунды. date.setMinutes(30, 0, 0)
- **setSeconds(seconds, [milliseconds]):** Устанавливает секунды и миллисекунды. date.setSeconds(0, 0)
- **setMilliseconds(milliseconds):** Устанавливает миллисекунды. date.setMilliseconds(0)
- **setTime(milliseconds):** Устанавливает время в миллисекундах с 1 января 1970 года. date.setTime(1627873200000)

## Преобразование даты в строку

- **toDateString():** Возвращает строку, содержащую только дату. date.toDateString()
- **toTimeString():** Возвращает строку, содержащую только время. date.toTimeString()
- **toLocaleDateString():** Возвращает строку с датой в локализованном формате. date.toLocaleDateString(locale, options)
- **toLocaleTimeString():** Возвращает строку с временем в локализованном формате. date.toLocaleTimeString(locale, options)
- **toLocaleString():** Возвращает строку с датой и временем в локализованном формате. date.toLocaleString(locale, options)
- **toISOString():** Возвращает строку в формате ISO 8601. date.toISOString()
- **toUTCString():** Возвращает строку с датой и временем в формате UTC. date.toUTCString()

- **toJSON():** Возвращает строку, представляющую дату, в формате JSON (тот же формат, что и toISOString). date.toJSON()

# Деструктуризация:

## Деструктуризация массивов

### Основной синтаксис:

```
const [a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

### Пропуск элементов:

```
const [a, , b] = [1, 2, 3];
console.log(a); // 1
console.log(b); // 3
```

### Остаточные элементы:

```
const [a, ...rest] = [1, 2, 3, 4];
console.log(a); // 1
console.log(rest); // [2, 3, 4]
```

### Значения по умолчанию:

```
const [a, b = 2] = [1];
console.log(a); // 1
console.log(b); // 2
```

## Деструктуризация объектов

### Основной синтаксис:

```
const { a, b } = { a: 1, b: 2 };
console.log(a); // 1
console.log(b); // 2
```

### Переименование переменных:

```
const { a: x, b: y } = { a: 1, b: 2 };
console.log(x); // 1
console.log(y); // 2
```

### Остаточные элементы:

```
const { a, ...rest } = { a: 1, b: 2, c: 3 };
console.log(a); // 1
console.log(rest); // { b: 2, c: 3 }
```

### Значения по умолчанию:

```
const { a, b = 2 } = { a: 1 };
console.log(a); // 1
console.log(b); // 2
```

## Деструктуризация в функциях

### Параметры функций:

```
function sum([a, b]) {
  return a + b;
}
console.log(sum([1, 2])); // 3
```

```
function multiply({ a, b }) {
  return a * b;
}
console.log(multiply({ a: 2, b: 3 })); // 6
```

### Значения по умолчанию в параметрах:

```
function greet({ name = "Guest" } = {}) {
  return `Hello, ${name}!`;
}
console.log(greet({ name: "Alice" })); // Hello, Alice!
console.log(greet()); // Hello, Guest!
```

## Примеры сложной деструктуризации

### Вложенные структуры:

```
const data = {
  id: 1,
  name: "John",
  address: {
    city: "New York",
    zip: "10001"
  }
};

const { name, address: { city, zip } } = data;
console.log(name); // John
console.log(city); // New York
console.log(zip); // 10001
```

### Комбинированные примеры:

```
const data = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

const [{ name: firstName }, { name: secondName }] = data;
console.log(firstName); // Alice
console.log(secondName); // Bob
```

## Вложенная деструктуризация

### Массивы в объектах:

```
const obj = { a: [1, 2, 3] };
const { a: [x, y] } = obj;
console.log(x); // 1
console.log(y); // 2
```

### Объекты в массивах:

```
const arr = [{ a: 1 }, { b: 2 }];
const [{ a }, { b }] = arr;
console.log(a); // 1
console.log(b); // 2
```

# Асинхронность

## Промисы (Promises)

**Промис (Promise)** — это объект, представляющий результат асинхронной операции, который может быть завершен (resolved) или отклонен (rejected).

```
const fetchData = fetch("https://cat-fact.herokuapp.com/facts")
  .then(response => {
    if (!response.ok) {
      throw new Error("Network response was not ok");
    }
    return response.json(); // Парсим ответ в JSON
  })
  .then(data => {
    console.log(data); // Выводим данные
  })
  .catch(error => {
    console.error("Error fetching data:", error); // Обработка ошибки
  })
  .finally(() => {
    console.log("Operation completed"); // Выполнится в любом случае
 });
```

**then()**: Вызывается при успешном завершении промиса. Принимает два аргумента: функции, которые будут вызваны при успешном завершении и при ошибке соответственно.

```
promise.then(onFulfilled, onRejected);
```

**catch()**: Вызывается при ошибке в промисе. Это сокращение для then(null, onRejected).

```
promise.then(onFulfilled, onRejected);
```

**finally()**: Вызывается в любом случае по завершении промиса (независимо от успешного завершения или ошибки).

```
promise.finally(onFinally);
```

## Цепочка промисов

```
fetchData
  .then(data => {
    console.log(data);
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve("Next data received");
      }, 1000);
    });
  })
  .then(nextData => {
    console.log(nextData); // "Next data received"
  })
  .catch(error => {
    console.error(error);
  })
  .finally(() => {
    console.log("Operation completed");
 });
```

# Асинхронность

## Async/ Await

**async и await** — это синтаксический сахар над промисами, который делает асинхронный код более читаемым и линейным

**async:** Объявляет функцию асинхронной, которая возвращает промис.

**await:** Ожидает завершения промиса и возвращает его результат. Может использоваться только внутри async функции.

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Ошибка:', error);
  } finally {
    console.log('Запрос завершён');
  }
}

fetchData();
```

## Обработка ошибок

Ошибки в асинхронных функциях обрабатываются с помощью конструкции

**try...catch**.

**finally** - сработает в любом случае

## Promise.all()

Используйте **Promise.all()** для одновременного выполнения нескольких промисов, а **finally** — для выполнения завершающего кода.

```
async function fetchMultipleData() {
  try {
    const [data1, data2] = await Promise.all([
      fetch('https://api.example.com/data1').then(res => res.json()),
      fetch('https://api.example.com/data2').then(res => res.json())
    ]);
    console.log(data1, data2);
  } catch (error) {
    console.error('Ошибка:', error);
  } finally {
    console.log('Все запросы завершены');
  }
}

fetchMultipleData();
```

# Замыкания

– Когда функция возвращает функцию

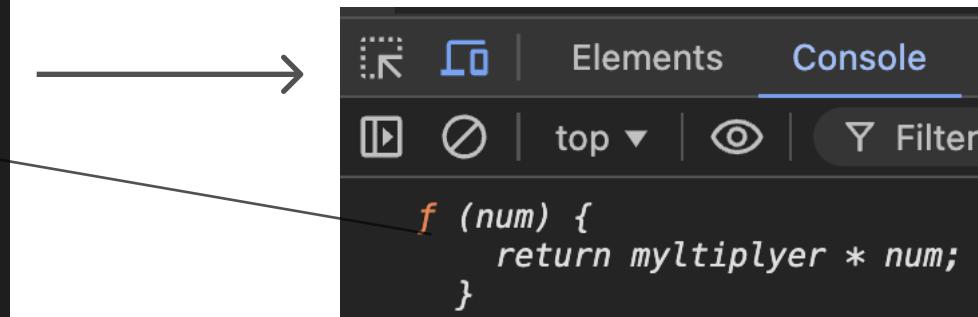
Пример вызова функции, которая возвращает другую функцию

```
38 function multiplyFn(myltipher) {  
39   return function (num) {  
40     ... return myltipher * num;  
41   };  
42 }  
43  
44 console.log(multiplyFn(2));
```



Консоль лог - для демонстрации результата

Получаем в консоль внутреннюю функцию. Логично



Цифра 2 пока просто для примера.  
Сейчас ни на что не влияет



## ЭТО И ЕСТЬ ЗАМЫКАНИЕ ФУНКЦИИ

Можно так.

```
38 function multiplyFn(myltipher) {  
39   ... return function (num) {  
40     ... return myltipher * num;  
41   };  
42 }  
43  
44 console.log(multiplyFn(4)(2)); // 8
```

Вызов функции myltplyFn(4)(2) сначала вызовет первую функцию с аргументом 4.

После вернет вторую функцию и сразу же ее вызовет с аргументом 2

Результат будет 8.  
 $4 * 2 = 8$

Памятка по областям видимости:



Замыкание возникает, когда внутренняя функция запоминает и имеет доступ к переменным из своей области видимости, даже после того, как внешняя функция, которая содержит эту внутреннюю функцию, завершает выполнение.

Когда внутренняя функция создается внутри внешней функции и использует переменные, объявленные в внешней функции, создается замыкание (сохранение внутренней функцией значений внешней).

МОЖНО ТАК  
ЗАМКНУТЬ

```
38 function multiplyFn(myltiple) {  
39   . . . return function (num) {  
40     . . . . return myltiple * num;  
41     . . . };  
42   }  
43  
44 console.log(multiplyFn(4)(2)); // 8
```

НО УДОБНЕЕ ТАК

```
38 function multiplyFn(myltiple) {  
39   . . . return function (num) {  
40     . . . . return myltiple * num;  
41     . . . };  
42   }  
43  
44 const multiplyByTwo = multiplyFn(2);  
45  
46 console.log(multiplyByTwo(7)); // 14
```

Создаем функциональное выражение  
(записываем функцию в переменную)

Вызываем функциональное выражение с  
аргументом 7

Замкнули(сохранили)  
параметр myltiple  
функции multiplyFn

Замкнули функцию multiplyFn(2) ее  
аргумент 2.

(Теперь всегда когда будет вызываться  
multiplyByTwo(число).  
Это число в аргументе multiplyByTwo будет  
умножаться на 2.

Когда внутренняя функция создается внутри внешней функции и использует  
переменные и параметры, объявленные в внешней функции, создается  
замыкание.

## ЗАМЫКАНИЕ ПО ШАГАМ

```
38 function multiplyFn(myltiple) {  
39   . . . return function (num) {  
40     . . . . return myltiple * num;  
41   . . . };  
42 }  
43  
44 const multiplyByTwo = multiplyFn(2);  
45  
46 console.log(multiplyByTwo(7)); // 14
```

1

Создали функцию multiplyFn с параметром (myltiple)

2

Функция multiplyFn делает одну вещь - создает и возвращает новую функцию (анонимную, без имени)

3

Создается функциональное выражение multiplyByTwo.

Значение переменной multiplyByTwo - это результат выполнения функции multiplyFn с аргументом 2.

А результат выполнения функции multiplyFn - это ее внутренняя функция.

Стало быть функциональное выражение multiplyByTwo - это анонимная внутренняя функция, функции multiplyFn. С сохраненным (замкнутым) значением параметра функции multiplyFn. Которая в данном примере является цифрой 2

4

Последнее. Мы запускаем функциональное выражение multiplyByTwo с параметром 7.

То есть теперь сработает внутренняя анонимная функция с аргументом 7.

Все что делает функция - это показывает (возвращает) результат своего аргумента (num) и аргумента внешней функции multiplyFn (умножает их друг на друга).

Получаем 14

## ЕЩЕ ОДИН ПРИМЕР ПО ШАГАМ

```
function outer() {  
    let count = 0; // Локальная переменная внешней функции  
    function inner() {  
        count++;  
        console.log(count);  
    }  
    return inner;  
}  
  
const myFunction = outer(); // myFunction становится замыканием  
myFunction(); // Выводит 1  
myFunction(); // Выводит 2
```

- outer объявляет переменную count и функцию inner.
- inner увеличивает count и выводит его. Хотя outer уже завершила выполнение после вызова myFunction = outer(), inner продолжает “видеть” count.
- inner сохраняет ссылку на лексическое окружение outer, что позволяет inner получать и изменять count.

# Памятка по формам

– Создание HTML форм

## Основная структура формы

- action: URL, на который будет отправлена форма.
- method: Метод HTTP для отправки формы (чаще всего GET или POST).

```
<form action="url_to_submit_form_data" method="POST">
  <!-- Элементы формы здесь -->
</form>
```

## Текстовые поля Элементы ввода

- text: Стандартное односторончное текстовое поле.
- password: Поле для ввода пароля, символы скрыты.
- email: Поле для электронной почты с встроенной валидацией.
- textarea: Многострочное текстовое поле.

```
<input type="text" name="username" required>
<input type="password" name="password" placeholder="Your Password">
<input type="email" name="email">
<textarea name="message" rows="4" cols="50"></textarea>
```

## Чекбоксы и радиокнопки

- checkbox: Чекбокс, может быть выбран или не выбран.
- radio: Радиокнопка, пользователи выбирают один из нескольких вариантов.

```
<input type="checkbox" name="subscribe" value="Yes" checked> Subscribe to newsletter<br>
<input type="radio" name="gender" value="male" checked> Male<br>
<input type="radio" name="gender" value="female"> Female
```

## Выпадающие списки и группы

- select и option: Выпадающий список с опциями на выбор.

```
<select name="car">
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

## Кнопки

- submit: Кнопка для отправки формы.
- button: Обычная кнопка, для использования с JavaScript.
- reset: Кнопка для сброса формы.

```
<button type="submit">Submit</button>
<button type="button" onclick="alert('Hello!')">Click Me</button>
<input type="reset" value="Reset">
```

## Скрытые поля

- hidden: Невидимое поле, которое не показывается пользователю, но содержит важные данные формы.

```
<input type="hidden" name="user_id" value="12345">
```

## Атрибуты для валидации

- required: Поле должно быть заполнено перед отправкой формы.
- pattern: Регулярное выражение, с которым должно совпадать значение поля.
- min и max: Минимальное и максимальное значения для числовых полей.
- maxlength и minlength: Максимальная и минимальная длина для текстовых полей.

```
<input type="text" name="username" required>
<input type="text" name="zip" pattern="[0-9]{5}">
<input type="number" name="age" min="18" max="99">
```

## Полезные практики

- Используйте label для улучшения доступности:

```
<label for="username">Username:</label>
<input type="text" id="username" name="username">
```

- Группируйте связанные элементы с помощью fieldset и legend:

```
<fieldset>
  <legend>Gender</legend>
  <input type="radio" name="gender" value="male" id="male"><label for="male">Male</label><br>
  <input type="radio" name="gender" value="female" id="female"><label for="female">Female</label>
</fieldset>
```

## Пример сложной формы

```
<form action="/submit-your-form-handler" method="POST">
  <fieldset>
    <legend>Personal Information</legend>
    <label for="firstname">First Name:</label>
    <input type="text" id="firstname" name="firstname" required><br><br>

    <label for="lastname">Last Name:</label>
    <input type="text" id="lastname" name="lastname" required><br><br>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required><br><br>

    <label for="dob">Date of Birth:</label>
    <input type="date" id="dob" name="dob" required><br><br>
  </fieldset>

  <fieldset>
    <legend>Account Details</legend>
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" minlength="5" required><br><br>

    <label for="password">Password:</label>
    <input type="password" id="password" name="password" minlength="8" required><br><br>

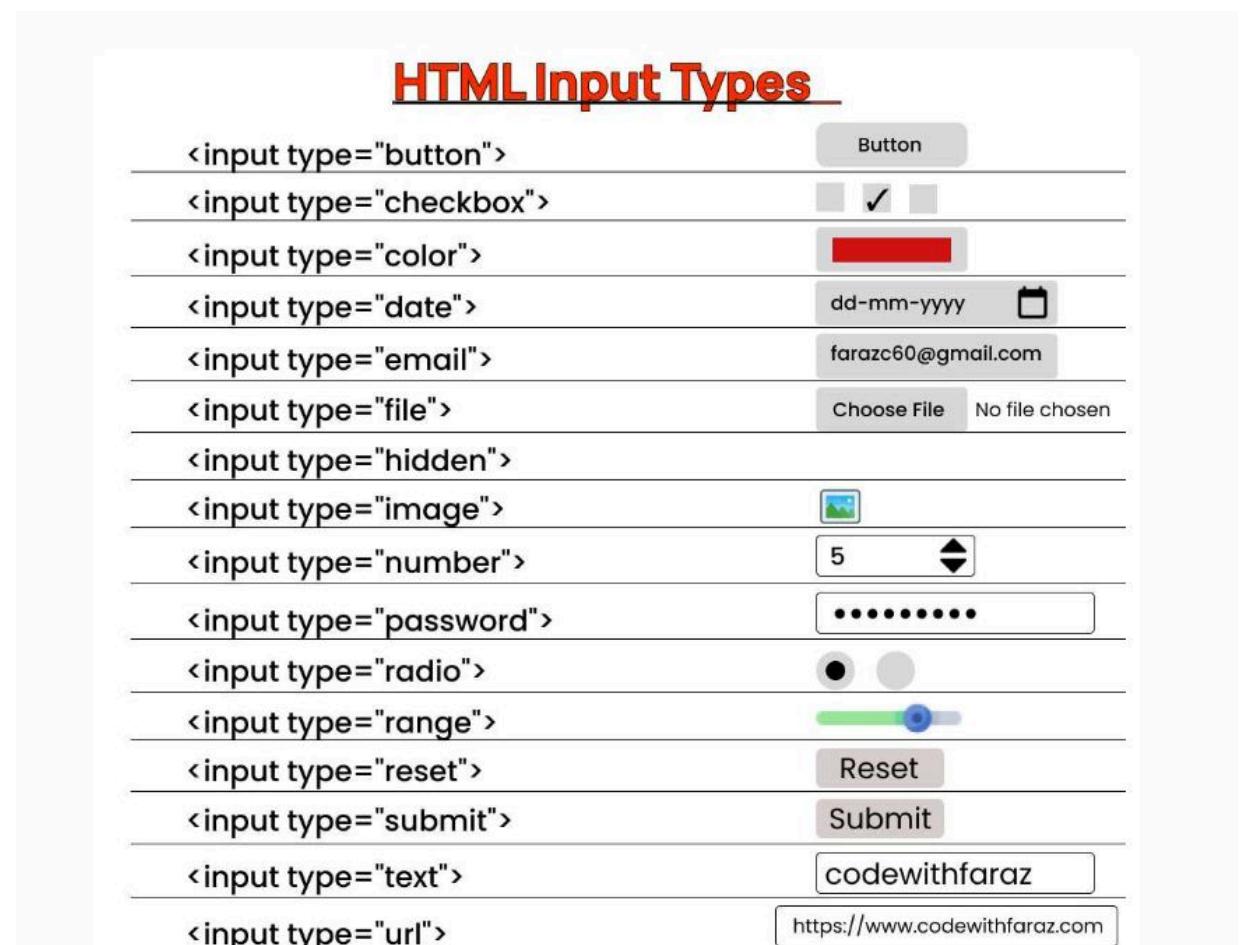
    <label for="confirm_password">Confirm Password:</label>
    <input type="password" id="confirm_password" name="confirm_password" minlength="8" required><br>
  </fieldset>

  <fieldset>
    <legend>Preferences</legend>
    <label>Favorite Color:</label>
    <select name="color" id="color">
      <option value="red">Red</option>
      <option value="green">Green</option>
      <option value="blue">Blue</option>
    </select><br><br>

    <label>Subscribe to newsletter:</label>
    <input type="checkbox" id="newsletter" name="newsletter" value="yes"><br><br>
  </fieldset>

  <button type="submit">Register</button>
  <button type="reset">Reset</button>
</form>
```

## Памятка по типам input



# Создаем новый проект React — vite

# Vite

— это современный инструмент сборки для проектов на JavaScript. Его основная цель — обеспечить быструю и эффективную разработку, особенно для современных фреймворков, таких как Vue, React, Svelte и другие.

## Как создать новый проект vite:

### 1. Откройте директорию

Откройте терминал и перейдите в папку, где хотите создать проект (терминал в VS Code или штатный).

### 2. Введите в терминале:

`npm create vite`

Задайте имя, выберите фреймворк и язык разработки.

### 3. Установите пакеты:

Откройте проект в VS Code и установите все необходимые пакеты командой в терминале:

`npm install`

### 4. Настройте ESLint (Опционально)

При необходимости настройте ESLint и скорректируйте другие настройки.

### 5. Удалите ненужные файлы и папки (Опционально)

При необходимости удалите ненужные файлы, папки, тестовый код.

### 6. Запустите проект

Ведите команду в терминале:

`npm run dev`

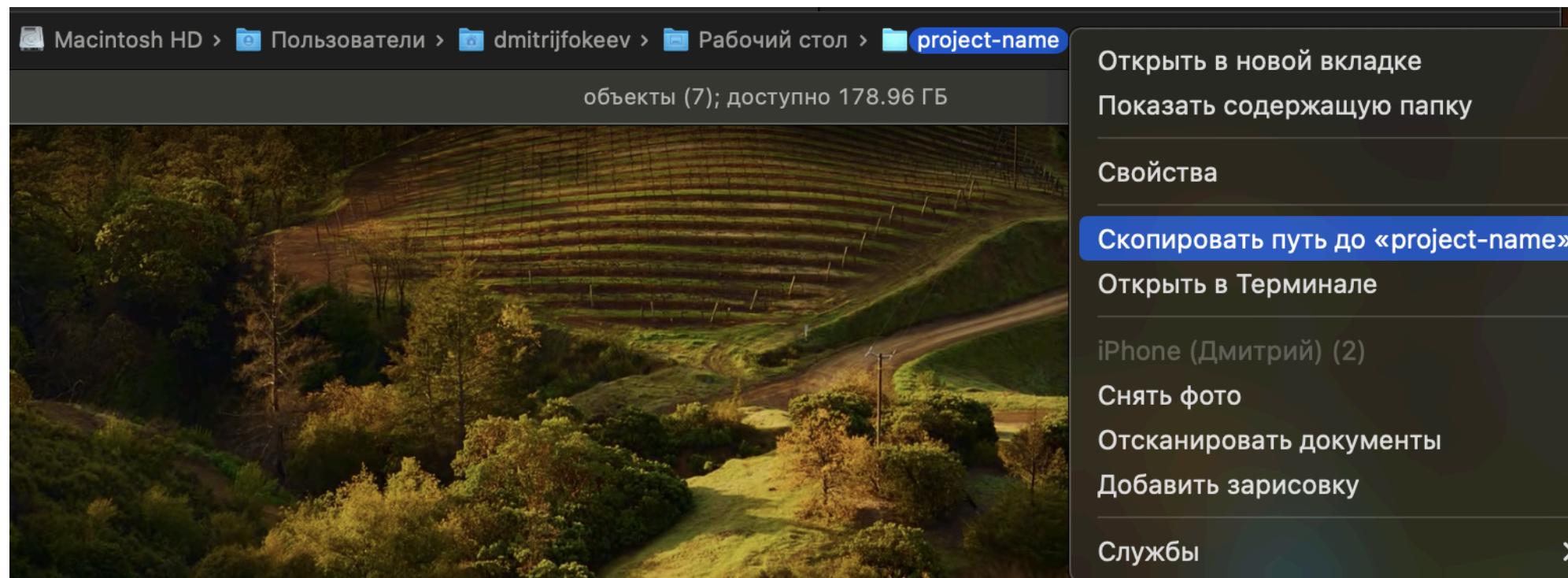
Скопируйте появившийся путь в адресную строку браузера.

### 7. Прервать проект

В терминале нажмите `ctr + C`

# Проблема с удалением или созданием файлов в проекте на mac:

Скопируйте путь до папки спроектом



И пропишите в терминал:

```
sudo chown -R $(whoami) /path/to/your/project-  
folder
```

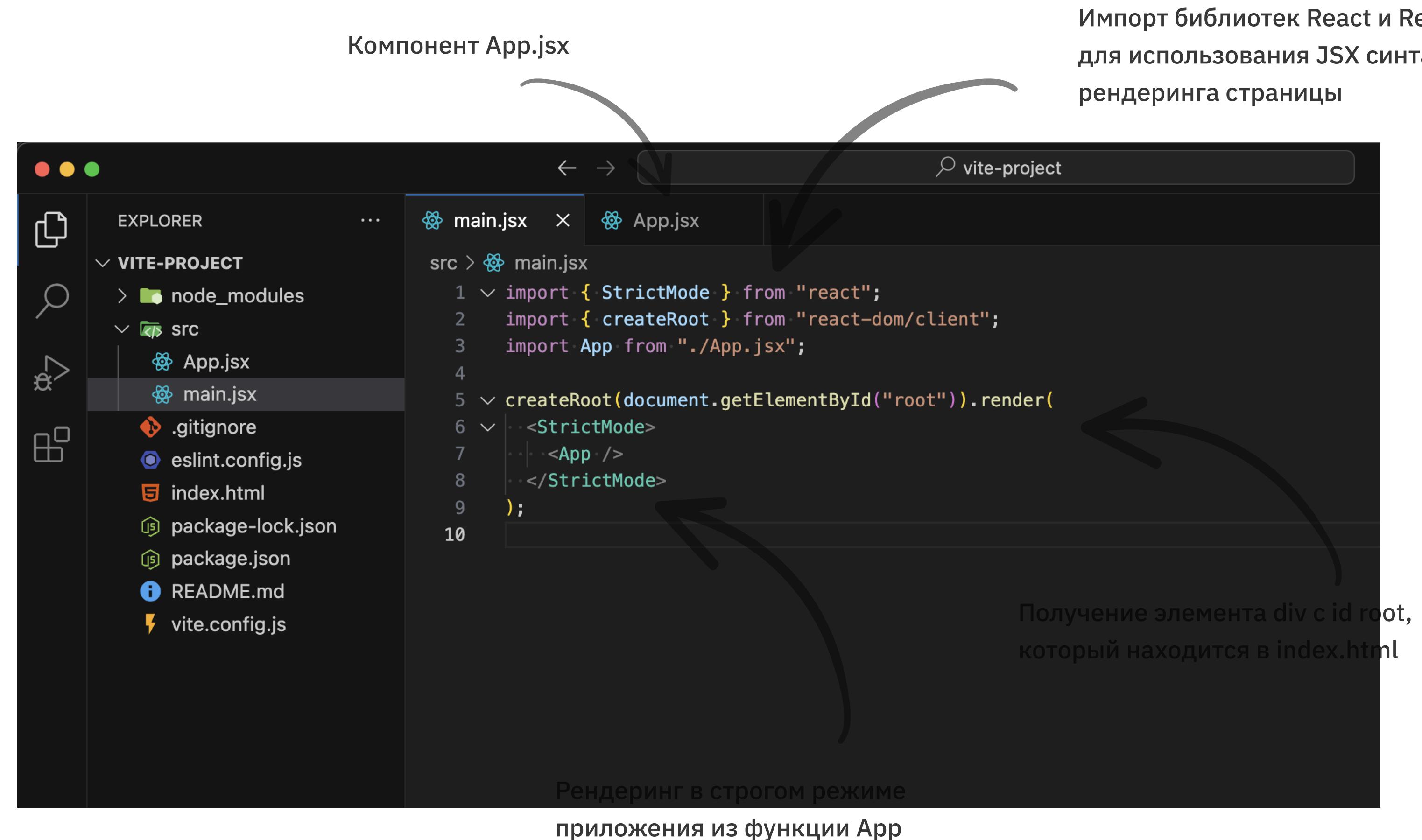
```
JS App.js x
src > JS App.js > App
1 import logo from './logo.svg';
2 import './App.css';
3
4 function App() {
5   return (
6     <div className="App">
7       <header className="App-header">
8         <img src={logo} className="App-logo" alt="logo" />
9         <p>
10           Edit <code>src/App.js</code> and save to reload.
11         </p>
12       </header>
13       <a
14         className="App-link"
15         href="https://reactjs.org"
16         target="_blank"
17         rel="noopener noreferrer"
18       >Learn React</a>
19     </div>
20   )
21 }
22
23 export default App;
```

Не удается записать файл  
"/Users/dmitrijfokeev/Desktop/project-name/src/index.js"  
(NoPermissions (FileSystemError): Error: EACCES: permission  
denied, open '/Users/dmitrijfokeev/Desktop/project-  
name/src/index.js')

Повторить

# Основная структура React приложения:

Компонент App.jsx



Импорт библиотек React и ReactDOM для использования JSX синтаксиса и рендеринга страницы

src > main.jsx

```
1 import { StrictMode } from "react";
2 import { createRoot } from "react-dom/client";
3 import App from "./App.jsx";
4
5 createRoot(document.getElementById("root")).render(
6   <StrictMode>
7     <App />
8   </StrictMode>
9 );
```

Получение элемента div с id root, который находится в index.html

Рендеринг в строгом режиме приложения из функции App

# **React.createElement**

– функция React

# React.createElement

— это функция, которую React использует для создания элементов.

## React.createElement

Эта функция является основой всей концепции React-элементов, и именно с ее помощью формируется структура, которую React затем использует для построения и обновления виртуального DOM.

Если бы у вас не было JSX, вы могли бы написать компонент таким образом:

```
function MyComponent() {
  return React.createElement(
    'div', // Тип элемента (HTML-тег 'div')
    { className: 'my-div' }, // Пропсы (здесь класс 'my-div')
    'Hello, world!' // Дочерний элемент (строка 'Hello, world!')
  );
}
```

Это эквивалентно следующему JSX-коду:

```
function MyComponent() {
  return (
    <div className="my-div">
      Hello, world!
    </div>
  );
}
```

Когда вы пишете JSX, такой как `<div>Hello, world!</div>`, он компилируется в вызов `React.createElement`. JSX — это синтаксический сахар, который упрощает запись React-элементов. Но под капотом React всегда использует `React.createElement`, чтобы создать объект элемента.

# React.createElement

— подробнее про работу

## Синтаксис React.createElement

```
React.createElement(type, props, ...children)
```

### Аргументы:

#### 1. type:

- Тип элемента, который нужно создать. Это может быть:
- Стока, которая указывает на HTML-тег (например, 'div', 'span').
- Функция или класс, если это React-компонент.

#### 2. props:

- Объект, содержащий все свойства (атрибуты) элемента. Если пропсов нет, можно передать null.
- Пример пропсов: { className: 'my-class', id: 'my-id' }.

#### 3. children:

- Дочерние элементы, которые могут быть переданы в элемент. Это может быть:
- Текстовый контент.
- Другие элементы React.
- Массив дочерних элементов.

## Пример с пропсами и дочерними элементами:

```
const element = React.createElement(
  'div', // Тип элемента
  { className: 'my-class', id: 'my-id' }, // Пропсы
  'Hello, ', // Первый дочерний элемент (текст)
  React.createElement('strong', null, 'world!') // Второй дочерний элемент – <strong>World!</strong>
);
```

Тоже самое что и:

```
<div className="my-class" id="my-id">
  Hello, <strong>world!</strong>
</div>
```

REACT JS

# JSX

– JavaScript XML

# JSX (JavaScript XML) — это расширение языка JavaScript

— Которое позволяет писать элементы,  
похожие на HTML или XML, прямо в коде  
React.

Проще говоря JSX - это синтаксис который  
позволяет писать HTML/CSS/JS в одном месте.

## Синтаксис JSX без JS

```
const element = <h1>Hello, world!</h1>;
```

## Синтаксис JSX с использованием JS

```
const name = 'World';
const element = <h1>Hello, {name}!</h1>;
```

## Пример создания React компонента с JSX

```
function UserProfile({ user }) {
  return (
    <div className="user-profile">
      {/* Используем фигурные скобки для встраивания JS выражений */}
      <h1>Welcome, {user.name}!</h1>
      <p>Age: {user.age}</p>
      {/* Условное отображение в JSX */}
      {user.age >= 18 ? (
        <p>You are an adult.</p>
      ) : (
        <p>You are under 18.</p>
      )}
    </div>
  );
}

// Пример использования компонента UserProfile
const user = {
  name: "Alice",
  age: 22
};

const element = <UserProfile user={user} />;
```

# JSX под капотом

— Или если бы мы использовали  
методы вместо JSX

“Под капотом” JSX трансформируется в вызовы React.createElement(). Например, JSX-код:

```
const element = <h1>Hello, world!</h1>;
```

Трансформируется в:

```
const element = React.createElement('h1', null, 'Hello, world!');
```

---

Функция React.createElement() принимает три аргумента:

1. Тип элемента (например, ‘h1’, ‘div’).
2. Объект с props (атрибутами), который может быть null или содержать свойства, такие как классы, стили и т. д.
3. Дети (children) элемента, которые могут быть текстом, другими элементами или выражениями.

REACT JS

# JSX – CSS

– Работа с CSS

## REACT JS

```
Компонент  
↓  
60  function Header() {  
61  |  return <h1 style={{ color: "red" }}>Header H1</h1>;  
62  }  
    ↓  
Объект со стилями.  
    ↓  
    Внимание на двойные фигурные скобки.
```

Стили записываются в виде объекта JavaScript, где ключи — это camelCase версии CSS-свойств, а значения — это строки или числа (для числовых значений, таких как width, padding и т.д., единица измерения по умолчанию — пиксели).

Значения цветов, пути и другие CSS-свойства, которые обычно записываются в кавычках, также должны быть строками.

## CSS в JSX

— Как работает CSS в компоненте

---

Можно создать отдельную переменную, значение которой будет объект со стилями

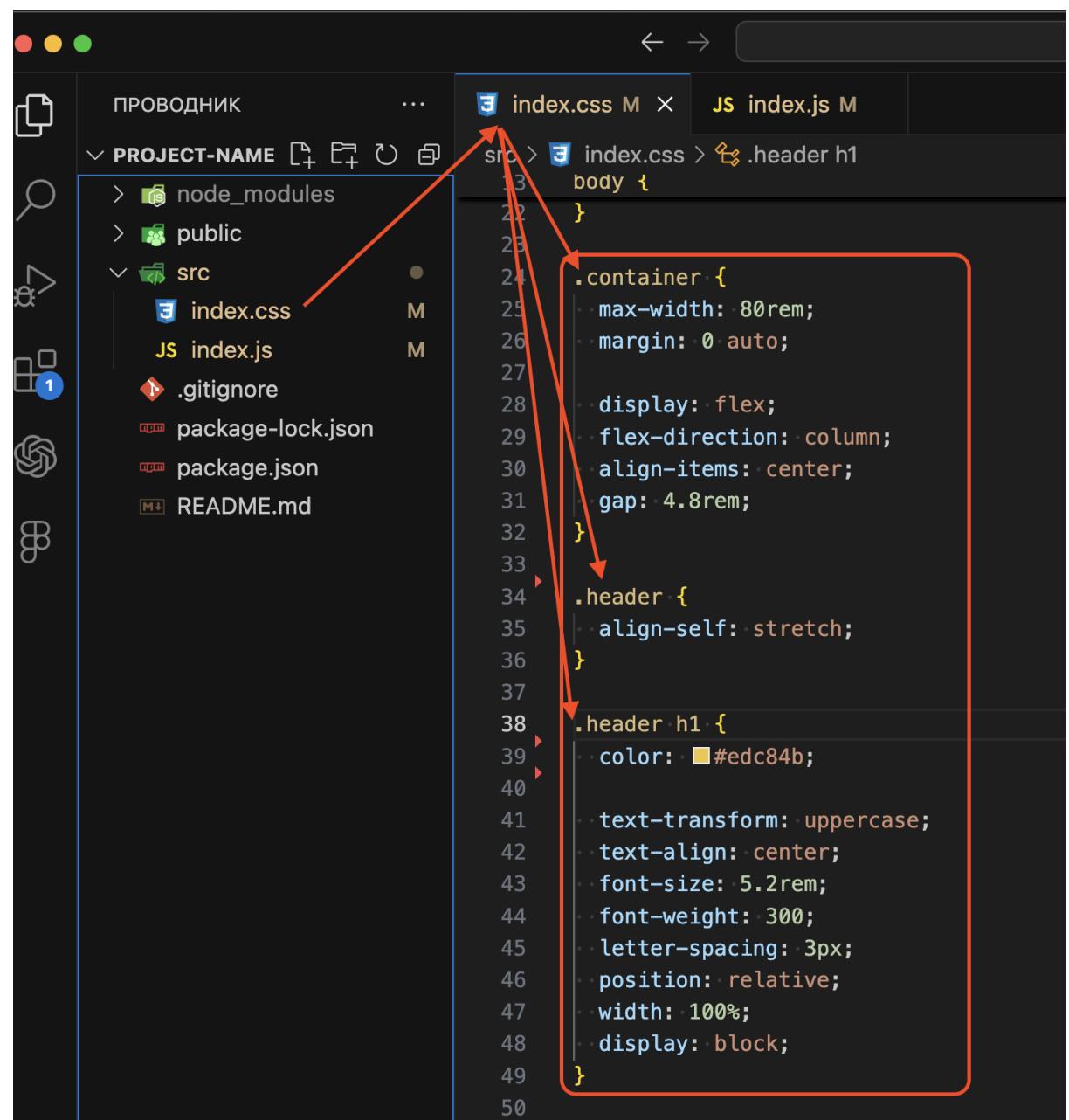
```
60  function Header() {  
61  |  const style = {  
62  |  |  color: "red",  
63  |  |  backgroundColor: "black",  
64  |  |  padding: 20, // Равносильно '20px'  
65  |  |  margin: "10px 5px 15px 20px",  
66  |  |  border: "1px solid blue",  
67  |  };  
68  |  |  return <h1 style={style}>Header H1</h1>;  
69  |  }  
70 }
```

REACT JS

# index.css

— Использование CSS файла с прописанными стилями

К определенным классам



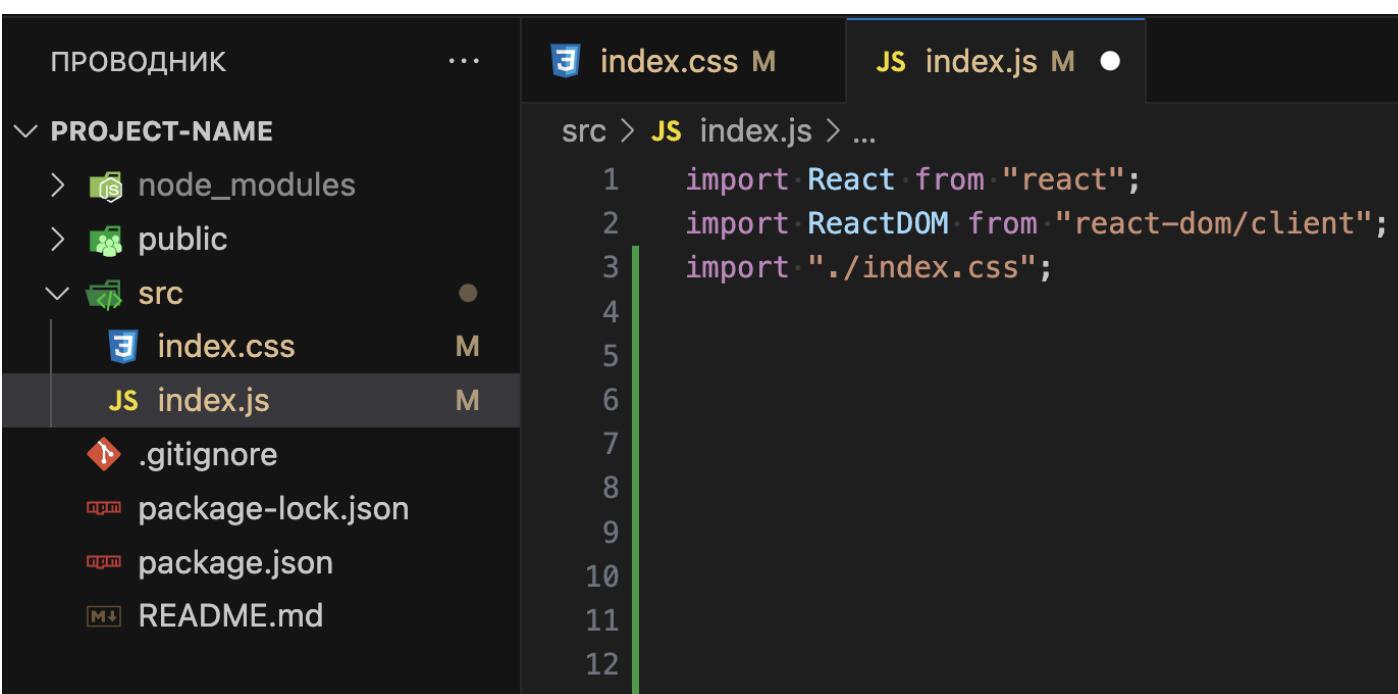
The screenshot shows a code editor interface with a dark theme. On the left is a sidebar titled 'ПРОВОДНИК' (File Explorer) showing the project structure:

- PROJECT-NAME
  - node\_modules
  - public
  - src
    - index.css M
    - index.js M
    - .gitignore
    - package-lock.json
    - package.json
    - README.md

The main editor area displays the content of 'index.css'. A red box highlights the following CSS code:

```
body {  
    margin: 0;  
}  
  
.container {  
    max-width: 80rem;  
    margin: 0 auto;  
}  
  
.header {  
    align-self: stretch;  
}  
  
.header h1 {  
    color: #edc84b;  
    text-transform: uppercase;  
    text-align: center;  
    font-size: 5.2rem;  
    font-weight: 300;  
    letter-spacing: 3px;  
    position: relative;  
    width: 100%;  
    display: block;  
}
```

Подключаем к файлу index.js файл index.css



The screenshot shows a code editor interface with a dark theme. The file 'index.js' is open. The sidebar on the left shows the project structure:

- PROJECT-NAME
  - node\_modules
  - public
  - src
    - index.css M
    - index.js M

The main editor area displays the content of 'index.js'. A green box highlights the following import statement:

```
import React from "react";  
import ReactDOM from "react-dom/client";  
import "./index.css";
```

CSS классы используются в JSX с использованием атрибута `className`, так как `class` является зарезервированным словом в JavaScript:

index.css file

```
38 .header {  
39     color: #edc84b;  
40     text-transform: uppercase;  
41     text-align: center;  
42     font-size: 5.2rem;  
43     font-weight: 300;  
44     letter-spacing: 3px;  
45     position: relative;  
46     width: 100%;  
47     display: block;  
48 }
```

index.js file

```
61     function Header() {  
62         return <h1 className="header">Header</h1>;  
63     }
```

# JSX - **памятка**

– В чем отличия от HTML / CSS / JS

## Общие правила JSX:

- Имена компонентов в React пишутся с большой буквы, чтобы отличать их от HTML-тегов.
- JSX работает почти как HTML, но мы можем вставлять «режим JavaScript» с помощью {}, чтобы добавлять ссылки на переменные, создание массивов или объектов, [].map(), тернарный оператор или любые другие JS выражения.
- Операторы (например, if, else, for, switch) не разрешены внутри JSX {}.
- Для условного отображения компонентов (вместо if else) используйте тернарный оператор, оператор &&.
- Комментарии в JSX: Внутри JSX-компонентов комментарии записываются как /\* комментарий \*/.

## Отличия между JSX и HTML

- Компоненты можно использовать внутри {}, так как они являются выражениями JavaScript.
- Инлайн стили CSS пишутся как style={object}
- Имена свойств CSS также пишутся в стиле camelCase.
- className вместо class в HTML.
- Каждый тег должен быть закрыт. Например: <img /> или <br />.
- Все обработчики событий и другие свойства должны быть в стиле camelCase. Например, onClick или onMouseOver.
- aria- и data- пишутся с дефисами, как и в HTML.

REACT JS

**пути в проектах**

React

— vite

## Относительный путь

— Относительные пути указывают на местоположение файла относительно текущей папки, используя

./ (в текущей папке)

ИЛИ

../ (на уровень выше)

## Абсолютный путь

— это полный адрес, который указывает на конкретное местоположение файла или ресурса, начиная с корня.

Абсолютный путь всегда полон и не зависит от текущего расположения файла.

(По умолчанию нет в проекте реакт)

## Особенности React

### 1. Абсолютный путь:

Если структура папок изменится, то пути не придется переделывать.  
В React можно использовать алиасы для указания корня.

### 2. Папка public:

Доступ к материалам из папки public можно получить без указания относительного пути к ней.  
Например, вместо ./public/img/some.jpg можно просто написать /img/some.jpg.

### 3. Экспорт модулей из src

Все файлы и компоненты в папке src нужно экспортировать из одного файла с помощью export или export default, чтобы импортировать их в другом файле.

## Экспорт:

Это процесс, который позволяет “вывести” часть кода (функцию, переменную или компонент) из текущего файла, чтобы другие файлы могли его импортировать.

## Виды экспорта:

### Экспорт по умолчанию (export default):

- Позволяет экспортировать один элемент как “главный” экспорт модуля.
- При импорте не требуется использовать фигурные скобки, и имя можно задавать любое.
- Обычно используется для экспортования основного компонента файла.

```
// Button.js
export default function Button() {
  return <button>Click me</button>;
}
```

### Именованный экспорт (export):

- Позволяет экспортировать несколько элементов из одного файла.
- При импорте требуются фигурные скобки, и имена должны совпадать с экспортруемыми.
- Часто используется для утилит, хелперов или если в модуле много экспортруемых значений.

```
// utils.js
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}
```

## Импорт:

Это процесс, который позволяет “взять” код, экспортированный из другого файла, и использовать его в текущем файле.

```
// App.js
import Button from './Button'; // Без фигурных скобок и с любым именем
```

```
// App.js
import { add, subtract } from './utils'; // В фигурных скобках, имя должно совпадать
```

# Props

– props (сокращение от properties)

# Props

— Способ передачи данных от родительских компонентов к дочерним.

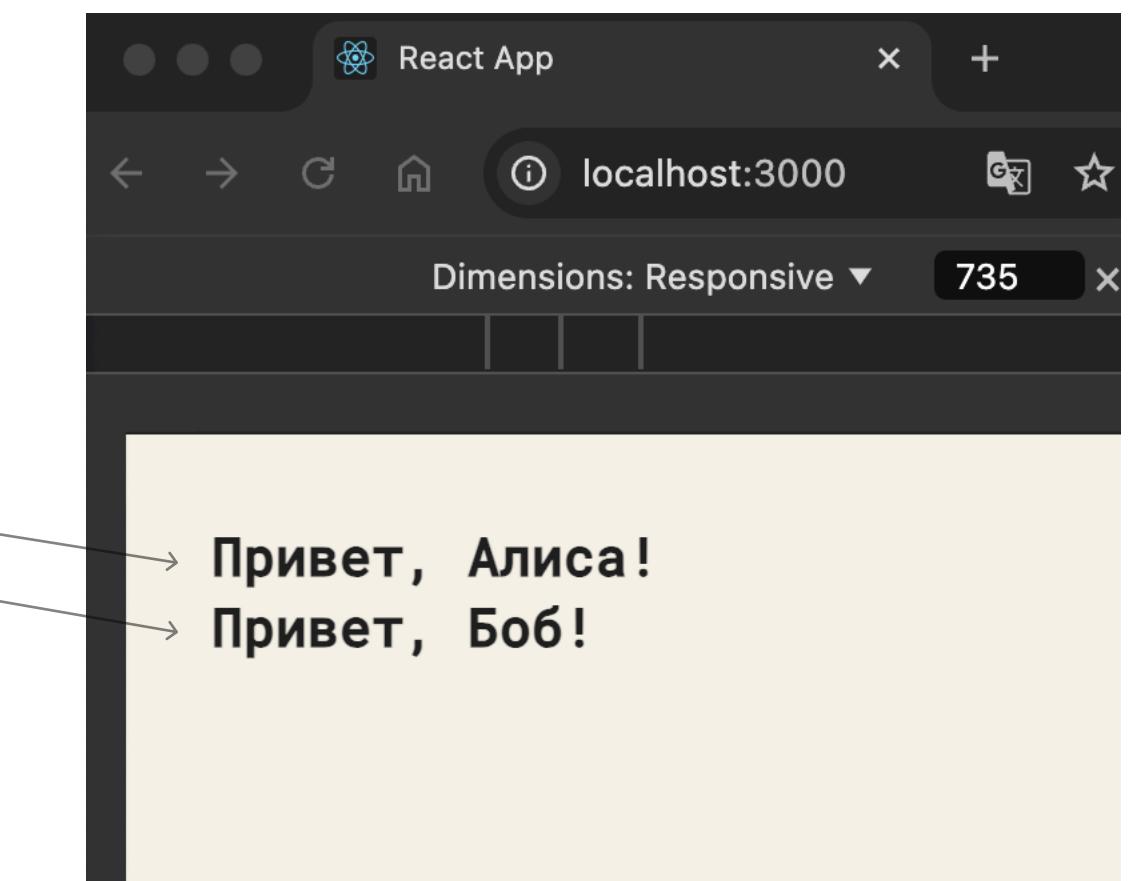
Props делает компоненты более переиспользуемыми и модульными. Компоненты получают props как параметр функции и используют их для отображения данных или для изменения своего поведения в зависимости от входных данных.

Родительский компонент, использующий в качестве дочернего, компонент <Welcome />

```
function App() {
  return (
    <div>
      <Welcome name="Алиса" />
      <Welcome name="Боб" />
    </div>
  );
}
```

Дочерний компонент

```
function Welcome(props) {
  return <h1>Привет, {props.name}!</h1>;
}
```



В данном случае, компонент App использует компонент Welcome дважды, передавая разные имена в качестве props. Каждый экземпляр компонента Welcome будет отображать приветствие с соответствующим именем.

# Props

— Способ передачи данных от родительских компонентов к дочерним.

## **Важные правила:**

Родительский компонент в котором используется дочерний компонент - это центр управления, который посылает команды какой именно должен быть элемент. С какими свойствами и в каком количестве

Дочерний компонент - это описание элемента(шаблон).

Какая у него структура и какие у него могут быть свойства.

В React данные передаются от родительского компонента к дочернему через props. Наоборот нельзя

Дочерний компонент получает данные как неизменяемые свойства (props), которые он может использовать, но не может изменять.

# map()

– map() в компонентах

## map()

— Конечно, вручную прописывать каждое свойство — не лучший вариант.

Поэтому, как правило, у нас имеется массив объектов с нужными данными.

Эти данные с помощью props мы и можем передать.

```
const products = [
  {
    id: 1,
    name: "Laptop",
    description: "High-performance laptop",
    price: 999,
    image: "https://example.com/laptop.jpg",
  },
  {
    id: 2,
    name: "Smartphone",
    description: "Latest model smartphone",
    price: 699,
    image: "https://example.com/smartphone.jpg",
  },
  {
    id: 3,
    name: "Headphones",
    description: "Noise cancelling headphones",
    price: 199,
    image: "https://example.com/headphones.jpg",
  },
];
```

REACT JS

# map()

Чтобы использовать данные из объектов в наших компонентах, нам нужно их перебрать обычным методом map()

```
const products = [
  {
    id: 1,
    name: "Laptop",
    description: "High-performance laptop",
    price: 999,
    image: "https://example.com/laptop.jpg",
  },
  {
    id: 2,
    name: "Smartphone",
    description: "Latest model smartphone",
    price: 699,
    image: "https://example.com/smartphone.jpg",
  },
  {
    id: 3,
    name: "Headphones",
    description: "Noise-cancelling headphones",
    price: 199,
    image: "https://example.com/headphones.jpg",
  },
];
```

```
function ProductList() {
  return (
    <div className="product-list">
      {products.map(product => (
        <ProductCard
          key={product.id}
          name={product.name}
          description={product.description}
          price={product.price}
          image={product.image}
        />
      ))}
    </div>
  );
}
```

1. Родительский компонент перебирает массив объектов методом map().
2. Каждый объект – это отдельный экземпляр компонента.
3. В каждый экземпляр компонента добавляется свойство, и в него передаются данные из объекта, такие как product.id, product.name и так далее.
4. Создан дочерний компонент, который принимает свойства (props) в параметрах функции. В этом компоненте описана структура и места использования этих свойств.

Тем самым, из “центра управления” (родительского элемента) указывается создать столько копий шаблона (дочернего компонента), сколько элементов в массиве, и использовать при этом данные из элементов массива в местах указанных в шаблоне (дочернем компоненте).

```
function ProductCard(props) {
  return (
    <div className="product-card">
      <img src={props.image} alt={props.name} style={{ width: '100%' }} />
      <h3>{props.name}</h3>
      <p>{props.description}</p>
      <span>Price: ${props.price}</span>
    </div>
  );
}
```

REACT JS

# map()

Немного упростим перебор в родительском компоненте, просто передав в дочерний объект целиком, а не отдельные его свойства.

```
const products = [
  {
    id: 1,
    name: "Laptop",
    description: "High performance laptop",
    price: 999,
    image: "https://example.com/laptop.jpg",
  },
  {
    id: 2,
    name: "Smartphone",
    description: "Latest model smartphone",
    price: 699,
    image: "https://example.com/smartphone.jpg",
  },
  {
    id: 3,
    name: "Headphones",
    description: "Noise cancelling headphones",
    price: 199,
    image: "https://example.com/headphones.jpg",
  },
];
```

```
function ProductList() {
  return (
    <div className="product-list">
      {products.map((product) => (
        <ProductCard productObj={product} />
      ))}
    </div>
  );
}
```

(Сравните с предыдущим слайдом  
изменения и поэкспериментируйте в  
коде)

```
function ProductCard(props) {
  return (
    <div className="product-card">
      <img src={props.productObj.image} alt={props.productObj.name} style={{ width: "100%" }} />
      <h3>{props.productObj.name}</h3>
      <p>{props.productObj.description}</p>
      <span>Price: ${props.productObj.price}</span>
    </div>
  );
}
```

# деструктуризация

props

– Сделаем еще проще и читабельней)

# Деструктуризация props

Вместо того, чтобы извлекать объект из переданных в компонент пропсов, мы можем сразу провести деструктуризацию пропсов в параметрах компонента.

## Версия без деструктуризации

```
function ProductList(props) {
  return (
    <div className="product-list">
      {props.products.map((product) => (
        <ProductCard product={product} />
      ))}
    </div>
  );
}

function ProductCard(props) {
  return (
    <div className="product-card">
      <img src={props.product.image} alt={props.product.name}>
      <h3>{props.product.name}</h3>
      <p>{props.product.description}</p>
      <span>Price: {props.product.price}</span>
    </div>
  );
}
```

## Версия с деструктуризацией

```
function ProductList({ products }) {
  return (
    <div className="product-list">
      {products.map((product) => (
        <ProductCard product={product} />
      ))}
    </div>
  );
}

function ProductCard({ product }) {
  return (
    <div className="product-card">
      <img src={product.image} alt={product.name}>
      <h3>{product.name}</h3>
      <p>{product.description}</p>
      <span>Price: {product.price}</span>
    </div>
  );
}
```

# Не присваиваем объект в параметрах компоненты

Причина, по которой в компонентах React обычно не указывают знак = в props, при деструктуризации заключается в том, что React автоматически обрабатывает пропсы и передает их в компонент.

Так проводить деструктуризацию не нужно

```
function ProductCard({ objectInProps } = props) {  
  ...  
  // Тело функции  
}
```

Нужно так

```
function ProductCard({ objectInProps }) {  
  ... // Тело функции  
}
```

Или так. Но прошлое удобнее и короче.

```
function ProductCard(props) {  
  const { objectInProps } = props;  
  ... // Тело функции  
}
```

REACT JS

# map()

Немного упростим перебор в родительском компоненте, просто передав в дочерний объект целиком, а не отдельные его свойства.

```
const products = [
  {
    id: 1,
    name: "Laptop",
    description: "High performance laptop",
    price: 999,
    image: "https://example.com/laptop.jpg",
  },
  {
    id: 2,
    name: "Smartphone",
    description: "Latest model smartphone",
    price: 699,
    image: "https://example.com/smartphone.jpg",
  },
  {
    id: 3,
    name: "Headphones",
    description: "Noise cancelling headphones",
    price: 199,
    image: "https://example.com/headphones.jpg",
  },
];
```

```
function ProductList() {
  return (
    <div className="product-list">
      {products.map((product) => (
        <ProductCard productObj={product} />
      ))}
    </div>
  );
}
```

(Сравните с предыдущим слайдом  
изменения и поэкспериментируйте в  
коде)

```
function ProductCard(props) {
  return (
    <div className="product-card">
      <img src={props.productObj.image} alt={props.productObj.name} style={{ width: "100%" }} />
      <h3>{props.productObj.name}</h3>
      <p>{props.productObj.description}</p>
      <span>Price: ${props.productObj.price}</span>
    </div>
  );
}
```

# Оператор **&&**

– рендеринг в зависимости от состояния

# Оператор &&

false && true) - Вернется первое false

(true && true) - Вернется второе true

Оператор && в React часто используется для условного рендеринга компонентов или элементов.

Это позволяет отобразить компонент или элемент только тогда, когда определённое условие истинно.

Оператор && возвращает первый операнд (левый), если он ложный (false, null, undefined, 0, NaN или пустая строка ""), и второй операнд (правый), если первый истинный (true).

```
// Компонент, отображающий сообщение об открытии
function OpenMessage() {
  return <h1>We are open! Come visit us.</h1>;
}

// Основной компонент приложения
function App() {
  const now = new Date(); // Получаем текущее время
  const hour = now.getHours(); // Извлекаем час из текущей даты

  // Устанавливаем условия отображения
  const isOpen = hour >= 9 && hour < 17; // Открыто с 9 до 17 часов

  return (
    <div>
      {isOpen && <OpenMessage />}
      {!isOpen && <h1>We are closed! See you tomorrow.</h1>}
    </div>
  );
}
```

Если сейчас 18 часов то isOpen это false.

Если isOpen = false то “We are Closed”

Если isOpen =true то “We are Open” (второй операнд)

Если перед и после && true, то вернется и отрендерится второе true.

Второе true компонент <OpenMessage />

# Тернарный оператор

– condition ? expressionIfTrue : expressionIfFalse

# Тернарный оператор

condition ? expressionIfTrue :  
expressionIfFalse

Тернарный оператор – это краткая форма условного ветвления if else.  
Если условие верно ? Выполни это : Если нет - это

```
const age = 18;  
const canVote = age >= 18 ? "Yes, you can vote." : "No, you cannot vote."  
console.log(canVote); // Выведет: "Yes, you can vote."
```

# Тернарный оператора в React

condition ? expressionIfTrue :  
expressionIfFalse

В JSX нельзя использовать ветвление if, так как JSX – это не JavaScript, а синтаксический сахар для вызовов функций, которые должны возвращать значения.

Однако вы можете использовать тернарный оператор внутри JSX:

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign up.</h1>}
    </div>
  );
}
```

## Использование if в React

Хотя if нельзя использовать непосредственно в JSX, вы можете использовать его в функции компонента перед возвратом JSX:

```
function UserProfile({ user }) {
  let message;

  if (user.isLoggedIn) {
    message = <h1>Welcome back, {user.name}!</h1>;
  } else {
    message = <h1>Please sign in.</h1>;
  }

  return (
    <div>
      {message}
      <p>Some other content...</p>
    </div>
  );
}
```

# Условный рендеринг

– if (true) {return some} else {return}

REACT JS

# Условный рендеринг

```
if (true) {  
    return some  
} else {  
    return another}
```

Объекты с данными

```
const products = [  
  {  
    id: 1,  
    name: "Laptop",  
    description: "High performance laptop",  
    price: 999,  
    image: "https://example.com/laptop.jpg",  
    isAvailable: true,  
  },  
  {  
    id: 2,  
    name: "Smartphone",  
    description: "Latest model smartphone",  
    price: 699,  
    image: "https://example.com/smartphone.jpg",  
    isAvailable: false,  
  },  
  {  
    id: 3,  
    name: "Headphones",  
    description: "Noise cancelling headphones",  
    price: 199,  
    image: "https://example.com/headphones.jpg",  
    isAvailable: true,  
  },  
];
```

Компоненты

```
// Компонент, который рендерит список товаров  
function ProductList({ products }) {  
  return (  
    <div className="product-list">  
      {products.map(product => <ProductCard product={product}>) }  
    </div>  
  );  
}  
  
function ProductCard({ product }) {  
  // Условный рендеринг на основе доступности товара  
  if (!product.isAvailable) {  
    return null; // Не рендерим карточку, если товар недоступен  
  }  
  
  return (  
    <div className="product-card">  
      <img src={product.image} alt={product.name} style={{ width: '100%' }} />  
      <h3>{product.name}</h3>  
      <p>{product.description}</p>  
      <span>Price: ${product.price}</span>  
    </div>  
  );  
}
```

ProductCard: Этот компонент получает объект product в качестве пропса.

В начале компонента есть условие, которое проверяет свойство isAvailable. Если товар недоступен (isAvailable равно false), функция возвращает null, что предотвращает рендеринг данной карточки товара.

ProductList: Компонент, который отображает список всех товаров.

Он использует ProductCard для отображения каждого товара, передавая в него данные о товаре.

<React.Fragment>

</React.Fragment>

Или

<>

</>

## REACT JS

```
<>
```

```
</>
```

React fragment позволяет группировать несколько элементов JSX, не создавая дополнительных DOM-узлов например не обернув все в дополнительный div

Вместо этого

```
function App() {  
  return (  
    <div>  
      <Header />  
      <Menu />  
      <Footer />  
    </div>  
  );  
}
```

Это

```
function App() {  
  return (  
    <>  
      <Header />  
      <Menu />  
      <Footer />  
    </>  
  );  
}
```

В верстке при этом не будет лишнего div

Либо так.

Одно и тоже

```
function App() {  
  return (  
    <React.Fragment>  
      <Header />  
      <Menu />  
      <Footer />  
    </React.Fragment>  
  );  
}
```

# Обработка событий

– onClick = {()=> alert('Click')}

## onClick = {function}

В React обработка событий, таких как клик по кнопке, реализуется с помощью атрибутов, начинающихся с on, таких как onClick

Отдельная функция которая будет срабатывать при событии(например клик)

```
function handleClick() {  
    alert('Кнопка была нажата!');  
}
```

Присваивание этой функции к атрибуту onClick компонента кнопки.  
В React это делается внутри JSX-кода:

```
<button onClick={handleClick}>  
    Нажми на меня  
</button>
```

Вы также можете использовать стрелочные функции для автоматической привязки контекста или определения обработчика прямо в JSX:

```
<button onClick={() => alert('Кнопка была нажата!')}>  
    Нажми на меня  
</button>
```

# Основные события

## Список основных событий

### События мыши

- onClick: Клик левой кнопкой мыши.
- onContextMenu: Клик правой кнопкой мыши для вызова контекстного меню.
- onDoubleClick: Двойной клик мышью.
- onDrag / onDragEnd / onDragEnter / onDragExit / onDragLeave / onDragOver / onDragStart / onDrop: События, связанные с перетаскиванием элементов.
- onMouseDown / onMouseEnter / onMouseLeave / onMouseMove / onMouseOut / onMouseOver / onMouseUp: События, связанные с действиями мыши.

### События клавиатуры

- onKeyDown / onKeyPress / onKeyUp: События нажатия клавиш на клавиатуре.

### События форм

- onChange: Изменение значения элемента формы (например, при вводе текста или выборе значения).
- onInput: Ввод данных пользователем в поле ввода.
- onSubmit: Отправка формы.

### События фокуса

- onFocus: Элемент получает фокус.
- onBlur: Элемент теряет фокус.

### События сенсорного ввода

- onTouchCancel / onTouchEnd / onTouchMove / onTouchStart: События, связанные с касаниями на сенсорных устройствах.

### События UI

- onScroll: Прокрутка элемента или страницы.

### События колеса мыши

- onWheel: Вращение колесика мыши.

### События изображения

- onLoad: Загрузка изображения завершена.
- onError: Ошибка при загрузке изображения или файла.

### Другие события

- onSelect: Выделение текста пользователем.
- onAnimationStart / onAnimationEnd / onAnimationIteration: События CSS анимаций.
- onTransitionEnd: Окончание CSS перехода.

# useState

– Управление состоянием

# useState

Это один из хуков в React, который позволяет компонентам иметь собственное состояние

Хук useState принимает начальное состояние и возвращает массив из двух элементов:

1. Текущее значение состояния.
2. Функцию для обновления этого состояния.

```
const [state, setState] = useState(initialState);
```

Пример:

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

function Counter() {
  // Инициализация состояния счётчика значением 0
  const [count, setCount] = useState(0);

  // Функция для увеличения счётчика
  function increment() {
    setCount(count + 1);
  }

  return (
    <div>
      <p>Вы нажали {count} раз</p>
      <button onClick={increment}>
        Нажми меня
      </button>
    </div>
  );
}
```

Деструктуризация

Функция setCount которая увеличивает состояние count на + 1

Функция изменения состояния срабатывает по нажатию на кнопку

useState(0) инициализирует count значением 0.

- setCount – это функция, которая позволяет обновить значение count. Вызов этой функции автоматически перерисует компонент с новым значением состояния.
- Кнопка в компоненте при клике вызывает функцию increment, которая увеличивает значение count на единицу.

# Изменение State

Изменение состояния можно делать только через `setState()`.

В React нельзя обновлять состояние напрямую, потому что React использует особый механизм для отслеживания изменений состояния, который основан на методе `setState`

## 1. Автоматическое управление рендерингом

React отслеживает, когда состояние изменяется через `setState`, и знает, что нужно выполнить рендер компонента. Если вы обновляете состояние напрямую, React не узнает об изменении и не выполнит рендеринг, а это приведет к тому, что интерфейс не отобразит актуальные данные.

## 2. Иммутабельность (неизменяемость) состояния

React опирается на неизменяемость состояния для повышения производительности. Когда состояние меняется через `setState`, React создает новую копию состояния, а не изменяет существующую. Это позволяет React быстро определить, что изменилось, сравнив новое и старое состояние.

## 3. Асинхронное обновление

Обновления состояния в React могут быть асинхронными, и они часто объединяются для повышения производительности. `setState` гарантирует, что React выполнит обновления последовательно и с актуальными данными.

Если состояние изменять напрямую, можно получить некорректные данные, так как React не будет знать об этих изменениях и не сможет правильно управлять очередью обновлений.

# useState **call-back**

– Используйте call-back когда изменяете текущее значение

# useState call-back

Используйте call-back для изменения состояния, когда новое состояние зависит от предыдущего

БЫЛО

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

function Counter() {
  // Инициализация состояния счётчика значением 0
  const [count, setCount] = useState(0);

  // Функция для увеличения счётчика
  function increment() {
    setCount(count + 1);
  }

  return (
    <div>
      <p>Вы нажали {count} раз</p>
      <button onClick={increment}>
        Нажми меня
      </button>
    </div>
  );
}
```

СТАЛО

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

function Counter() {
  const [count, setCount] = useState(0);

  function increment() {
    setCount((currentCount) => currentCount + 1);
  }

  return (
    <div>
      <p>Вы нажали {count} раз</p>
      <button onClick={increment}>
        Нажми меня
      </button>
    </div>
  );
}
```

Когда вы используете callback форму `setCount((count) => count + 1)`, вы передаете функцию, которая получает текущее значение состояния в качестве аргумента и возвращает новое значение.

Это гарантирует, что вы работаете с самым последним значением состояния на момент выполнения обновления, что особенно важно в случаях, когда состояние может измениться несколько раз в короткие временные промежутки

# Когда использовать **useState** – И как именно

# useState

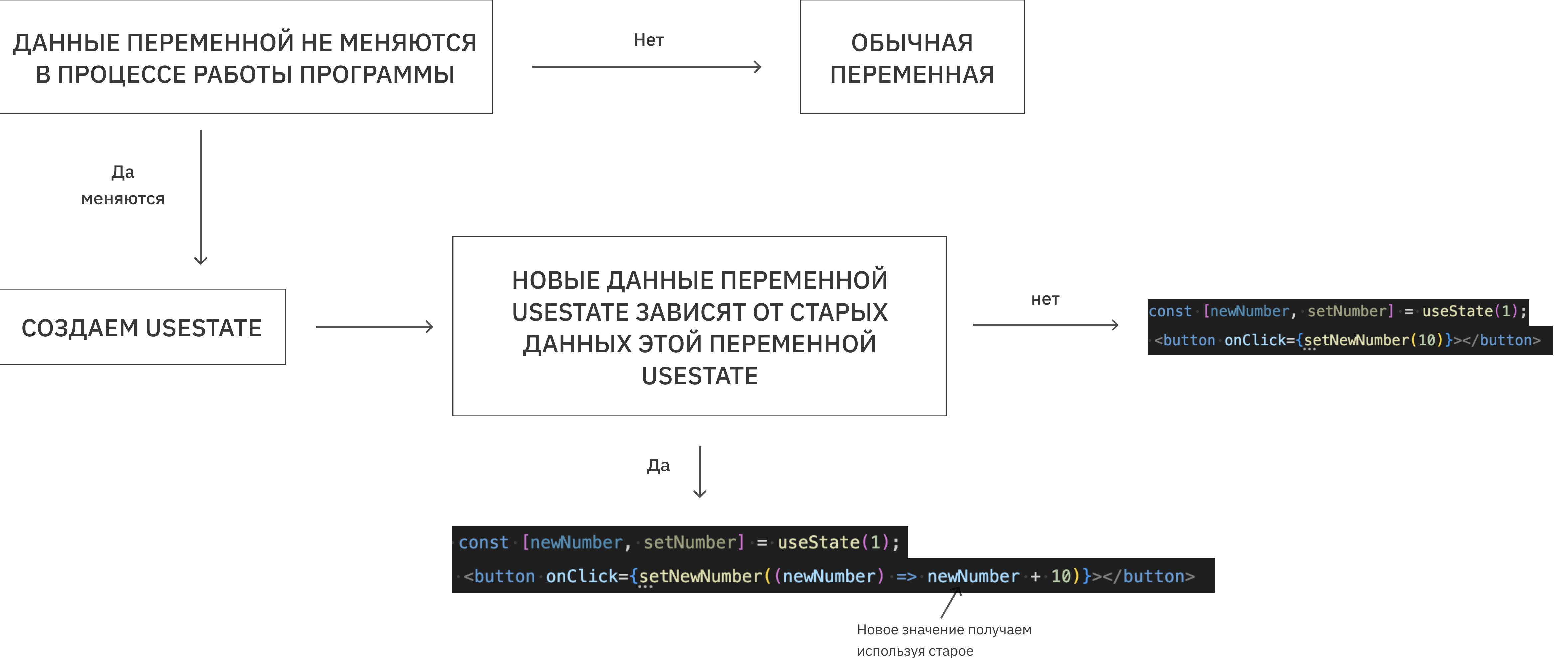
Когда использовать

## Что делает useState

1. Сохраняет значение между рендерами: Значение, созданное с помощью useState, сохраняется при каждом повторном рендере компонента.
2. Вызываетrerендеринг при изменении: При обновлении состояния через setState компонент автоматически перерисовывается, чтобы отобразить новые данные.
3. Сохраняет неизменность состояния: Обновление состояния через setState создает новую версию состояния, позволяя React сравнить старое и новое значения для оптимизации рендеринга.

## Используйте useState когда нужно

1. Отслеживание состояния для рендеринга: Если значение, созданное через useState, меняется, компонент автоматически рендерится заново, отображая новое значение в интерфейсе.
2. Сохранение значений между рендерингами: Переменные, объявленные через useState, сохраняют свое значение между рендерингами, в отличие от обычных переменных, которые сбрасываются.



# React

# декларативный

– Но что это значит?

# Декларативный или императивный

- Декларативный код описывает что должно произойти и больше сосредоточен на конечном результате.
- Императивный код описывает как это сделать, сосредоточен на пошаговом процессе.

## Декларативный и императивный

— это два подхода к написанию кода, которые описывают разные стили взаимодействия с программами. Эти стили часто упоминаются в контексте программирования и разработки интерфейсов, особенно в React и других современных библиотеках.

### 1. Декларативный

Декларативный стиль фокусируется на том, что программа должна сделать, а не на том, как это сделать. В декларативном коде мы описываем желаемый конечный результат, и система сама решает, как его достичь. Это подход более высокого уровня абстракции, при котором упрощается написание и понимание кода, так как он ближе к естественному языку.

В React, при помощи JSX, мы пишем декларативный код для описания того, как должен выглядеть пользовательский интерфейс:

```
function App() {  
  return <h1>Hello, World!</h1>;  
}
```

### 2. Императивный подход

Императивный стиль фокусируется на том, как программа должна выполнить задачу. Здесь разработчик дает точные инструкции системе, описывая пошагово процесс выполнения. Это более низкий уровень абстракции, где разработчик управляет конкретными операциями.

#### Пример: чистый JavaScript

Если мы хотим вывести текст “Hello, World!” на экран в чистом JavaScript, это может выглядеть так:

```
const heading = document.createElement('h1');  
heading.textContent = 'Hello, World!';  
document.body.appendChild(heading);
```

# Array.from()

– Создание массива с помощью Array.from()

## Автоматически

```
Array.from({ length: 20 }, (_, i) => i + 1);
```

Позволяет создать новый массив из итерируемого или подобного массиву объекта

## Вручную

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20];
```

### Метод Array.from() имеет несколько параметров:

**1. Первый параметр** – объект, который похож на массив. Этот объект не содержит реальных значений, но имеет свойство `length`, установленное в 20, что позволяет `Array.from` определить, сколько элементов должно быть в результирующем массиве.

**2. Второй параметр** – это функция отображения (`map function`), которая вызывается для каждого индекса нового массива. Эта функция принимает два аргумента:

- `_` (или любое другое имя, которое вы предпочитаете) – значение элемента в массиве, которое в данном случае не используется, потому что исходные элементы фактически отсутствуют.
- `i` – индекс текущего элемента массива.

### Как это работает в вашем коде

Функция отображения в коде `(_, i) => i + 1` создает значения для каждого элемента нового массива.

Она начинается с `i`, который является индексом каждого элемента, начиная с 0 до 19 (так как индексы в JavaScript начинаются с 0). Прибавляя 1 к каждому индексу, вы получаете числа от 1 до 20.

# props.children

– В React, `children` – это специальный пропс, который позволяет передавать дочерние элементы внутрь компонента.

# props.children

children – это обычный пропс, который передается в компонент. Его содержимое – это всё, что находится внутри открывающего и закрывающего тегов компонента.

В этом примере `<h2>` и `<p>` передаются как пропс `children` в компонент `Card`. Этот пропс мы можем использовать в JSX внутри самого компонента.

Таким образом, всё, что мы поместим между открывающимся и закрывающимся тегами компонента при его использовании, будет передано в пропс `children`.

```
function Card({ children }) {
  return <div className="card">{children}</div>;
}

// Использование компонента Card
function App() {
  return (
    <Card>
      <h2>Заголовок</h2>
      <p>Текст внутри карточки</p>
    </Card>
  );
}
```

КЛЮЧЕВОЕ СЛОВО CHILDREN ЗАРЕЗЕРВИРОВАНО!

## children может быть:

**Примитивом:** строка, число, логическое значение  
(например, "Hello", 123, true).

- **Элементом React:** один элемент или несколько элементов.
- **Функцией:** например, для реализации render props.
- **Массивом:** список элементов.

Можно задать значение children по умолчанию:

```
function DefaultContainer({ children = <p>Контент по умолчанию</p> }) {  
  return <div>{children}</div>;  
}
```

# props. без значения

– Всегда true

## props.без значения

В React, если вы передаёте проп (например, `isDisabled`) без значения, он автоматически считается равным `true`.

Это стандартное поведение JavaScript и JSX, которое делает код более лаконичным.

### Как это работает

Когда вы пишете:

```
<Button variant="primary" isDisabled>  
  Disabled  
</Button>
```

ЭТО ЭКВИВАЛЕНТНО следующему:

```
<Button variant="primary" isDisabled={true}>  
  Disabled  
</Button>
```

# Component Composition

– Композиция компонентов

# Component Composition

— это техника, при которой компоненты объединяются друг с другом для создания более сложных пользовательских интерфейсов

Создаем компонент Modal который можно переиспользовать с разным контентом внутри,  
(Контент будет зависеть от prop.children)

```
function Modal({ isOpen, onClose, children }) {
  return (
    <div onClick={onClose}>
      <div>
        <button onClick={onClose}>
          &times;
        </button>
        {children}
      </div>
    </div>
  );
}
```

```
function App() {
  const [isOpen1, setIsOpen1] = useState(false);
  const [isOpen2, setIsOpen2] = useState(false);

  return (
    <div className="App">
      <button onClick={() => setIsOpen1(true)}>Открыть Модальное Окно 1</button>
      <button onClick={() => setIsOpen2(true)}>Открыть Модальное Окно 2</button>

      <Modal isOpen={isOpen1} onClose={() => setIsOpen1(false)}>
        <ModalContent1 />
      </Modal>

      <Modal isOpen={isOpen2} onClose={() => setIsOpen2(false)}>
        <ModalContent2 />
      </Modal>
    </div>
  );
}
```

2 разных компонента, со своим уникальным контентом, для использования в компоненте “модальное окно”

```
function ModalContent1() {
  return (
    <div>
      <h2>Контент Модального Окна 1</h2>
      <p>Это контент для первого случая.</p>
    </div>
  );
}

function ModalContent2() {
  return (
    <div>
      <h2>Контент Модального Окна 2</h2>
      <p>Это контент для второго случая.</p>
    </div>
  );
}
```

Как избежать props  
drilling  
— с помощью props.children

# Как избежать prop drilling

— с помощью `props.children`

Этот подход позволяет избежать передачи данных через промежуточные компоненты. Вместо этого мы просто передаем нужные компоненты как `children` в `NavBar`.

```
export default function App() {
  const [people] = useState([
    { id: 1, name: "John Doe", age: 28 },
    { id: 2, name: "Jane Smith", age: 34 },
    { id: 3, name: "Bob Johnson", age: 45 },
  ]);

  return (
    <NavBar>
      <Logo />
      <NumResult people={people} />
    </NavBar>
  );
}

function NavBar({ children }) {
  return <nav className="nav-bar">{children}</nav>;
}

function Logo() {
  return (
    <div className="logo">
      <h1>People Finder</h1>
    </div>
  );
}

function NumResult({ people }) {
  return (
    <p className="num-results">
      Found <strong>{people.length}</strong> people
    </p>
  );
}
```

**явная передача**

компонента

— вместо `props.children`

## 2 подхода:

### Использование props.children

```
function Layout({ children }) {
  return <div className="layout">{children}</div>;
}

function App() {
  return (
    <Layout>
      <Header />
      <Content />
      <Footer />
    </Layout>
  );
}
```

#### Плюсы:

- Гибкость: Позволяет передавать произвольное количество и типы дочерних элементов. Это удобно для компонентов, которые выступают в роли контейнеров или оболочек (например, Card, Modal, Layout).
- Композиция компонентов: props.children особенно полезен, когда нужно рендерить любые вложенные компоненты внутри другого компонента.
- Чистый код: Часто более читабельный код, когда компонент служит “контейнером” для других компонентов.

#### Минусы:

- Неявность: Если вы передаете компоненты через props.children, сложнее контролировать и понимать, что именно передается внутрь, особенно если компоненты передаются из нескольких мест.
- Не подходит для специфических случаев: Если требуется строгий контроль над количеством или типом дочерних компонентов, использование props.children может быть менее удобным.

## Использование явных props для передачи компонентов

```
function Layout({ header, content, footer }) {
  return (
    <div className="layout">
      {header}
      {content}
      {footer}
    </div>
  );
}

function App() {
  return (
    <Layout
      header={<Header />}
      content={<Content />}
      footer={<Footer />}
    />
  );
}
```

Плюсы:

**Явность:** Явная передача компонентов через пропсы делает интерфейс вашего компонента более предсказуемым и легко читаемым. Вы точно знаете, что и где будет рендериться.

- **Лучший контроль:** Легче контролировать количество и типы передаваемых компонентов.
- **Упрощает типизацию:** В TypeScript и PropTypes легче описать, что именно ожидается в пропсах.

Минусы:

- **Меньшая гибкость:** Если нужно передавать произвольное количество компонентов или создавать сложные композиции, явные пропсы могут оказаться слишком ограничивающими.
- **Избыточность:** Когда компонент принимает много компонентов как пропсы, код может становиться громоздким и сложным для чтения.

# useEffect

– Hook

## useEffect

— используется для управления побочными эффектами (side effects) в функциональных компонентах.

### **useEffect**

Позволяет выполнять код, который должен запускаться после рендера компонента, а также управлять побочными эффектами, которые происходят из-за изменения состояния или пропсов.

# useEffect

— Как работает?

## useEffect

useEffect запускается после каждого рендера компонента.

Это означает, что он работает как при монтировании (первый рендер), так и при обновлении компонента (ререндер).

Можно передать массив зависимостей вторым аргументом в useEffect.

Если массив зависимостей пустой ([]), useEffect выполняется только при монтировании компонента.

Если в массив зависимостей включены определённые значения, useEffect будет запускаться только при изменении этих значений.

```
function DataFetchingComponent() {
  const [data, setData] = useState([]);

  useEffect(() => {
    // Запрос данных – побочный эффект
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));

    // Если нужен очистительный эффект (например, отписка от событий):
    return () => {
      console.log('Очистка перед размонтированием');
    };
  }, []); // Пустой массив зависимостей – выполняется только один раз при монтировании

  return (
    <div>
      {data.map(item => (
        <div key={item.id}>{item.name}</div>
      ))}
    </div>
  );
}
```

**Размонтирование:** Возвращаемая функция из useEffect используется для очистки побочных эффектов (например, отписка от событий или остановка таймеров).

# useEffect

– dependency array и подробнее про процесс работы

# useEffect

## — Массив зависимостей

### Массив зависимостей в useEffect

Определяет в каком именно случае нужно запустить код из useEffect

```
useEffect(() => {
  async function fetchData() {
    const response = await fetch('https://api.example.com/data');
    const result = await response.json();
    setData(result); // Обновление состояния
  }

  fetchData();
}, []); // Пустой массив зависимостей – эффект выполнится только при первом рендеринге
```

Переменные в массиве зависимостей являются своего рода обработчиками событий, для запуска кода в useEffect.

Если массив зависимостей пустой, код внутри useEffect выполнится только один раз — после первого рендера компонента и его отрисовки на экране. Если в массиве указаны различные состояния (state), то useEffect будет выполняться как при первоначальном рендеринге, так и после каждого обновления этих состояний.

#### 1. Первый рендеринг:

- Компонент рендерится впервые, React выполняет весь код компонента, включая вычисления JSX, чтение состояний через useState и т.д. На этом этапе useEffect не выполняется.
- После завершения первого рендеринга React отображает результат на экране.

#### 2. Выполнение useEffect:

- После того как рендер завершился и изменения отображены на экране, React начинает выполнять эффекты, указанные в useEffect. Если в эффекте есть асинхронный код, например, запрос к API, этот запрос отправляется, и эффект продолжает выполняться.
- Данные, полученные от API, обрабатываются, и через setState обновляется состояние компонента.

#### 3. Повторный рендеринг после изменения состояния:

- Когда состояние изменяется с помощью setState, React запускает повторный рендеринг компонента.
- В процессе повторного рендеринга React снова выполняет всю функцию компонента, включая вычисления JSX.

Однако эффекты (useEffect) на этом этапе не выполняются повторно если у useEffect установлен пустой массив зависимостей

#### 4. Повторное выполнение useEffect (при наличии состояний в массиве зависимостей):

- После завершения второго рендеринга React снова выполняет эффекты, но только если изменились зависимости, указанные в массиве зависимостей useEffect.

# Side Effects

– Побочные эффекты

# Side Effect

— (побочные эффекты) — это любые операции, которые происходят вне текущего контекста выполнения функции или компонента.

Действия, которые изменяют внешнее состояние, взаимодействуют с внешними системами или имеют последствия, выходящие за рамки функции.

## Side Effects (Побочные эффекты)?

-это любое действие, которое выходит за пределы текущей функции и взаимодействует с внешним миром или изменяет состояние, которое находится вне скоупа этой функции.

В контексте React, побочные эффекты включают:

- Запрос данных из API.
- Изменение заголовка страницы.
- Подписка на события, такие как изменение размера окна.
- Установка таймеров или интервалов.
- Чтение и запись в локальное хранилище (localStorage или sessionStorage).
- Получение текущей геолокации пользователя.
- Показ системных уведомлений.
- Подключение к WebSocket для получения данных в реальном времени.
- Изменение глобальных переменных или состояний, доступных за пределами функции.

# Отписка от событий

## Отписываемся от всего что срабатывает не однократно

### 1. Таймеры и интервалы

- Почему: Они продолжают работать после удаления компонента.
- Решение: Используйте clearTimeout или clearInterval.

```
return () => clearTimeout(timer);
```

### 2. Слушатели событий

- Почему: Обработчики событий, добавленные через addEventListener, остаются активными после удаления компонента.
- Решение: Используйте removeEventListener.

```
return () => window.removeEventListener("resize", handleResize);
```

### 3. WebSocket

- Почему: Соединение остаётся открытым и продолжает передавать данные.
- Решение: Закройте соединение.

```
return () => socket.close();
```

### 4. Подписки на данные

- Почему: Подписки (например, RxJS) продолжают получать данные после удаления компонента.
- Решение: Отмените подписку.

```
return () => subscription.unsubscribe();
```

### 5. Анимации

- Почему: Запросы через requestAnimationFrame продолжаются.
- Решение: Используйте cancelAnimationFrame.

```
return () => cancelAnimationFrame(frameId);
```

### 6. Геолокация

- Почему: Отслеживание через watchPosition продолжается.
- Решение: Используйте clearWatch.

```
return () => navigator.geolocation.clearWatch(watchId);
```

### 7. Асинхронные операции

- Почему: Попытка обновить состояние после завершения асинхронной операции вызовет ошибку.
- Решение: Используйте флаг для отмены.

```
return () => {  
  isMounted = false;  
};
```

### 8. Внешние API или сторонние библиотеки

- Почему: Они могут продолжать использовать ресурсы или данные.
- Решение: Следуйте документации для отмены операций.

```
return () => subscription.unsubscribe();
```

### 9. Браузерные события

- Почему: Прокрутка, клавиатурные или другие глобальные события остаются активными.
- Решение: Удалите обработчики.

```
return () => window.removeEventListener("scroll", handleScroll);
```

# Race conditions

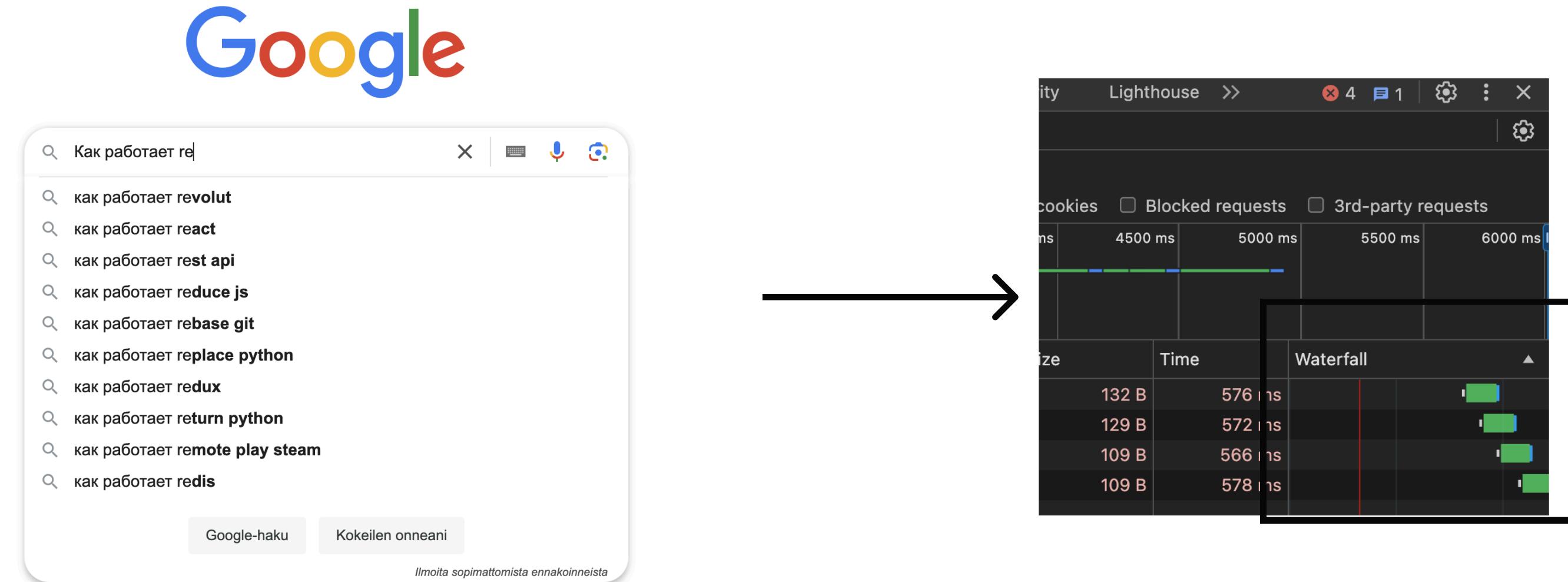
— гонка состояний

REACT JS

## race conditions

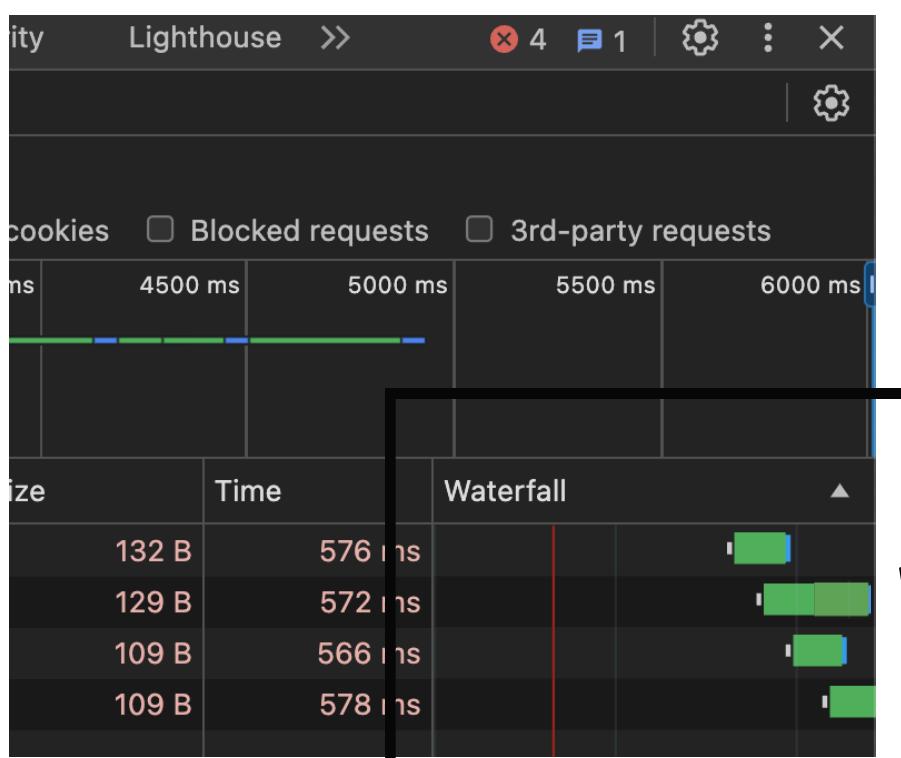
— ГОНКА СОСТОЯНИЙ

После каждого нажатия на клавишу, создается новый запрос на получение данных, и данные на экране обновляются



Если на какой-то букве запрос будет идти дольше, чем на последней, то этот застрявший запрос дойдет последним.

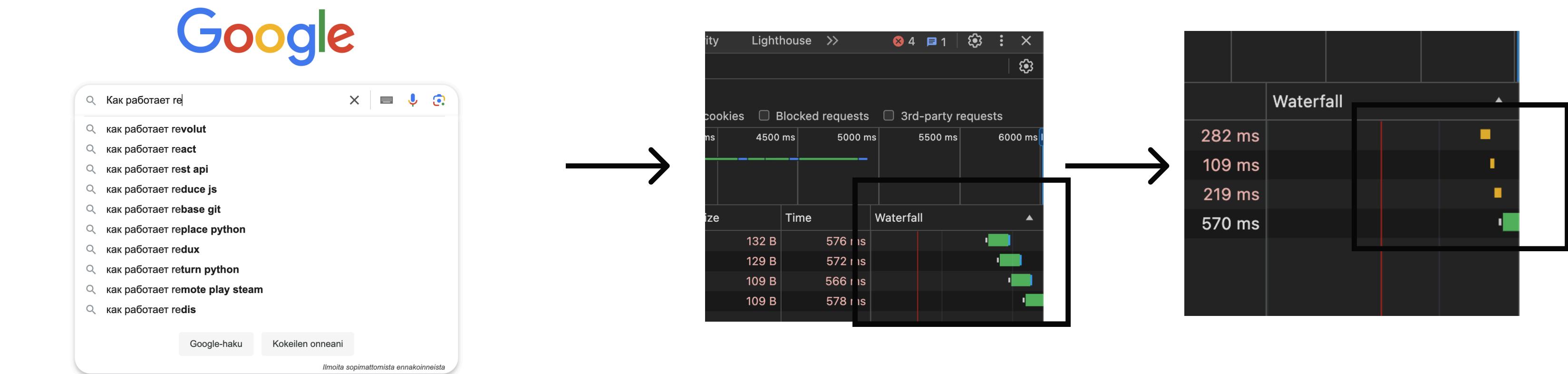
И, соответственно, его результат выпадет на экран вместо того, который мы напечатали последним



**Чтобы все работало корректно**, при создании нового запроса нам сначала нужно завершить предыдущий

## race conditions

— Гонка состояний



## В этом нам поможет AbortController

**AbortController** создаёт объект, который можно использовать для отмены одного или нескольких асинхронных операций. Он используется вместе с `fetch`, чтобы прервать запрос, если больше нет необходимости в его выполнении (например, если пользователь переключился на другой экран или если запрос был заменён другим).

```
function MyComponent() {
  useEffect(() => {
    const controller = new AbortController(); // Создаем новый AbortController
    const signal = controller.signal; // Получаем сигнал из контроллера

    async function fetchData() {
      try {
        const response = await fetch('https://api.example.com/data', { signal });
        const data = await response.json();
        console.log(data);
      } catch (err) {
        if (err.name === 'AbortError') {
          console.log('Запрос был отменен');
        } else {
          console.error('Произошла ошибка:', err);
        }
      }
    }

    fetchData();

    return () => controller.abort(); // Прерываем запрос при размонтировании или повторном рендеринге
  }, []);
}
```

— Можете скопировать код из  
комментариев ниже и протестировать

# race conditions — подробнее

— Можете скопировать код из  
комментариев ниже и протестировать

## AbortController это встроенный в JavaScript API объект

- 1. Создание AbortController:** Внутри useEffect создаётся новый экземпляр AbortController, который предоставляет сигнал (signal) для управления отменой запроса.
- 2. Передача signal в fetch:** В параметрах fetch передаётся signal, что позволяет отменить запрос, если AbortController вызовет abort().
- 3. Отмена предыдущего запроса:** В useEffect возвращается функция очистки, которая вызывает controller.abort(). Это отменяет любой предыдущий запрос, который всё ещё может быть в процессе выполнения, если query изменяется до завершения запроса.
- 4. Обработка ошибки отмены:** Если запрос был отменён, возникает ошибка типа AbortError. Мы проверяем её в блоке catch, чтобы не отображать сообщение об ошибке пользователю, если запрос был отменён преднамеренно.

```
function MyComponent() {
  useEffect(() => {
    const controller = new AbortController(); // Создаем новый AbortController
    const signal = controller.signal; // Получаем сигнал из контроллера

    async function fetchData() {
      try {
        const response = await fetch('https://api.example.com/data', { signal });
        const data = await response.json();
        console.log(data);
      } catch (err) {
        if (err.name === 'AbortError') {
          console.log('Запрос был отменен');
        } else {
          console.error('Произошла ошибка:', err);
        }
      }
    }

    fetchData();

    return () => controller.abort(); // Прерываем запрос при размонтировании или повторном рендеринге
  }, []); // Пустой массив зависимостей — эффект выполняется один раз при монтировании
}
```

# **useEffect vs Events**

— Что выбрать?

# useEffect vs Events

— Что выбрать для работы с side effects?

## Используйте useEffect, когда вам нужна автоматизация.

Используйте useEffect когда side effect (например, запрос к серверу) должен происходить самостоятельно при каких-то условиях работы программы.

Например, раз в определённое время или при изменении тех или иных state.

Тот момент, когда должен происходить side effect (например, запрос к серверу), определяется в dependency array (Массив зависимостей).

Так же, useEffect позволяет выполнить функцию при размонтировании компонента (его удалении).

## Events c side effects

Используйте в 3-х случаях:

1. Когда выполнение side effect (например, запрос к серверу) нужно только после события, выполненного пользователем (onClick, onChange и так далее).
2. Когда не нужно выполнять side effect (например, запрос к серверу) при первой загрузке приложения (при монтировании компонента).
3. Когда не нужно выполнять функцию отчистки

# Классовые vs функциональные компоненты

— и их отличия

# Класс VS Функция

— Классовые и функциональные компоненты — это два способа создания компонентов в React

— Можете скопировать код из комментариев ниже и протестировать

## Классовые компоненты

Классовые компоненты в React были основным способом создания компонентов до введения хуков в версии React 16.8 (январь 2019 года). До этого функциональные компоненты использовались только для простых задач, так как они не могли иметь состояние или методы жизненного цикла.

```
import React, { Component } from 'react';

class MyClassComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export default MyClassComponent;
```

### Основные особенности классовых компонентов:

- Состояние: В классовых компонентах состояние хранится в объекте state, который можно изменять с помощью метода setState.
- Методы жизненного цикла: Классовые компоненты используют специальные методы, такие как componentDidMount, componentDidUpdate, componentWillUnmount, для выполнения действий на различных этапах жизненного цикла компонента.
- Экземпляры: Для каждого классового компонента React создает экземпляр класса, который хранит состояние и методы.

# Класс VS Функция

— Классовые и функциональные компоненты — это два способа создания компонентов в React

— Можете скопировать код из комментариев ниже и протестировать

## Функциональные компоненты

Функциональные компоненты в React — это более простой способ создания компонентов. Изначально они были “бездумными” (stateless), то есть не могли управлять состоянием и не имели методов жизненного цикла. Однако с введением хуков в версии React 16.8 функциональные компоненты стали более мощными и теперь могут выполнять все те же задачи, что и классовые компоненты.

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

function MyFunctionComponent() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default MyFunctionComponent;
```

### Основные особенности функциональных компонентов:

- Простота: Функциональные компоненты проще по синтаксису и читаемости. Они представляют собой обычные функции, которые принимают пропсы и возвращают JSX.
- Хуки: Функциональные компоненты используют хуки, такие как useState, useEffect, useContext, чтобы управлять состоянием и выполнять побочные эффекты.
- Отсутствие методов жизненного цикла: В функциональных компонентах нет методов жизненного цикла, как в классовых. Вместо этого хуки, такие как useEffect, выполняют похожие задачи.

# Класс VS Функция

— Классовые и функциональные компоненты — это два способа создания компонентов в React

Функциональные компоненты с введением хуков стали предпочтительным способом создания компонентов в React. Они предлагают более простой и понятный синтаксис, а также позволяют легко управлять состоянием и побочными эффектами.

Однако классовые компоненты все еще актуальны и могут использоваться в старых проектах или в тех случаях, где они предпочтительны.

## Основные отличия

Классовые компоненты	Функциональные компоненты
Используют синтаксис классов ES6	Используют синтаксис функций
Управляют состоянием через <code>this.state</code>	Управляют состоянием через хуки ( <code>useState</code> )
Используют методы жизненного цикла	Используют хуки ( <code>useEffect</code> )
Экземпляры класса создаются при рендере	Нет экземпляров (компонент — это просто функция)
Более сложный и многословный синтаксис	Более простой и лаконичный синтаксис

# Components, instances, elements

— и их отличия

# Components – (Компоненты)

**Компонент** – это фундаментальная часть приложения React.

Компоненты представляют собой функции, которые принимают входные данные (пропсы) и возвращают элементы React, описывающие, как должна выглядеть часть пользовательского интерфейса.

```
function MyComponent(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

Компоненты – это всего лишь шаблоны, которые описывают, как должна выглядеть определенная часть интерфейса.

# Instances – (Экземпляры)

Экземпляры компонентов относятся к классовым компонентам и создаются при рендеринге таких компонентов.

Каждый раз, когда React рендерит классовый компонент, он создает новый экземпляр этого класса.

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { count: 0 };  
  }  
  
  render() {  
    return <h1>Count: {this.state.count}</h1>;  
  }  
}
```

При каждом рендрере создается новый экземпляр компонента MyComponent, который хранит свое состояние (`this.state`) и методы жизненного цикла.

**Важно:** Функциональные компоненты в React не имеют экземпляров, потому что они не являются классами.

**НО**

Для простоты и общего понимания, многие называют экземпляром вызов функционального компонента.

# Elements — (Элементы)

**Элемент** – это простейшая единица React. Элементы представляют собой описание того, что должно быть отрендерено на экране. Они не содержат логики или состояния, как компоненты, и не имеют методов жизненного цикла. Элементы React являются неизменяемыми объектами.

```
function MyComponent() {  
  return <div>Hello, world!</div>;  
}
```

Функциональный  
компонент

Элемент

Проще говоря.

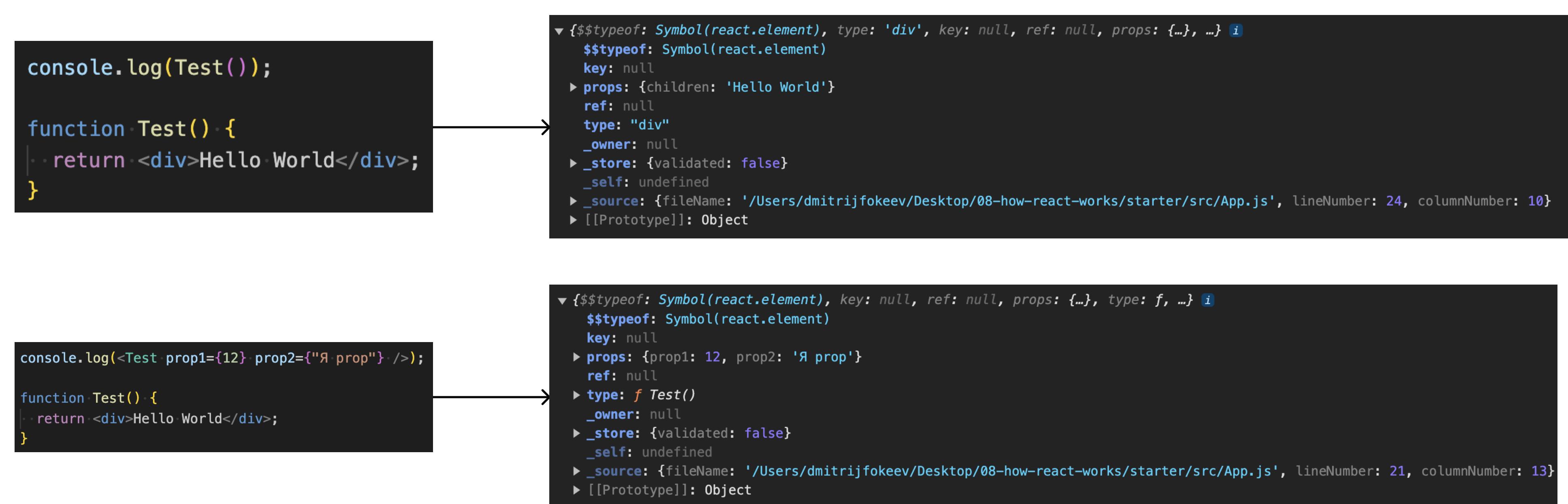
Элемент - это то что возвращает компонент.

# Element

— Это объект

Элемент в React — это объект, который React использует для описания того, что нужно отрендерить в DOM.

Когда вы пишете JSX, такой как `<div>Hello, world!</div>`, этот JSX компилируется в вызов функции `React.createElement`, которая создает объект элемента.



## Основные различия в верхних 2-х вызовах:

- HTML-тег vs. Компонент: В первом случае элемент представляет собой стандартный HTML-тег ('div'), а во втором случае — это React-компонент (Test).
- Рендеринг: В первом случае React рендерит HTML-тег, во втором — вызывает компонент Test, чтобы получить результат его рендеринга.

# Rendering

– Как работает

## Рендеринг компонентов в React

— это процесс, при котором React вызывает функцию компонента, чтобы получить описание интерфейса в виде React-элементов.

Эти элементы используются для создания и обновления виртуального DOM, который затем синхронизируется с реальным DOM.

**Рендеринг** - это не отрисовка на экране, а отрисовка под капотом React

**Внутренний рендеринг:** Когда мы говорим о рендеринге в React, мы часто имеем в виду процесс вызова компонента, который возвращает элементы React (через JSX). Эти элементы формируют виртуальный DOM — внутреннее представление интерфейса, с которым работает React.

**Отрисовка:** После рендеринга React сравнивает новый виртуальный DOM с предыдущим, чтобы определить минимальные изменения, которые нужно внести в реальный DOM (это процесс называется “диффинг”). Только после этого изменения синхронизируются с реальным DOM, что и приводит к видимым обновлениям на экране.

### Рендеринг включает:

1. Вызов компонента и получение элементов.
2. Построение/обновление виртуального DOM.
3. Сравнение (диффинг) виртуального DOM с предыдущей версией.

А вот отрисовка на экране (синхронизация с реальным DOM) — это уже следующий шаг, который React выполняет после рендеринга.

Технически он называется *Commit phase*

# Виртуальный дом

— Как выглядит?

```
function App() {
  const [showModal, setShowModal] = useState(true);

  return (
    <div className="app">
      <SomeComponent />
      {showModal && (
        <ModalWindow>
          <Overlay>
            <h3>Оставьте отзыв</h3>
            <button>5 звезд!</button>
          </Overlay>
        </ModalWindow>
      )}
      <Btn>{showModal ? "Оценить" : "Закрыть окно"}</Btn>
    </div>
  );
}
```

```
{
  type: "div",
  props: {
    className: "app",
    children: [
      {
        type: SomeComponent,
        props: {}
      },
      showModal
        ? {
            type: ModalWindow,
            props: {
              children: {
                type: Overlay,
                props: {
                  children: [
                    {
                      type: "h3",
                      props: {
                        children: "Оставьте отзыв"
                      }
                    },
                    {
                      type: "button",
                      props: {
                        children: "5 звезд!"
                      }
                    }
                  ]
                }
              }
            }
          : null,
        {
          type: Btn,
          props: {
            children: showModal ? "Оценить" : "Закрыть окно"
          }
        }
      ]
    ]
  }
}
```

# Rendering

— Когда запускается рендеринг?

## **Рендеринг компонентов запускается в двух случаях:**

### **Первичный рендеринг (Initial rendering):**

Происходит, когда компонент впервые добавляется в DOM.

В этот момент React вызывает компонент, чтобы получить его JSX (или элемент), а затем строит виртуальный DOM, который синхронизируется с реальным DOM.

### **Перерендеринг (Rerender):**

Происходит каждый раз, когда изменяются пропсы или состояние компонента.

React вызывает компонент снова, сравнивает новый виртуальный DOM с предыдущим (это называется “диффинг”) и обновляет реальный DOM только там, где произошли изменения (*Если изменения действительно были*).

# Виртуальный DOM

– Как работает

# Виртуальный DOM

— это объект содержащий дерево компонентов и элементов.

## Virtual DOM

- это легковесная, абстрактная копия реального DOM, которая существует в памяти и используется для управления интерфейсом.

Виртуальный DOM представляет собой дерево объектов, где каждый объект соответствует узлу интерфейса, например, элементу HTML или компоненту React.

## Как работает виртуальный DOM?

### 1. Создание Virtual DOM:

- Когда компонент рендерится впервые, React создаёт Virtual DOM — описание интерфейса в виде дерева элементов.
- Это дерево создаётся на основе JSX, который возвращает компонент, и служит входными данными для построения структуры Fiber Tree.

### 2. Обновление и сравнение (diffing):

- При изменении состояния (state) или пропсов (props) React создаёт новое Virtual DOM-дерево.
- Это дерево используется для обновления “рабочего дерева” (work-in-progress Fiber Tree), которое сравнивается с текущим состоянием (current Fiber Tree).
- React выявляет минимальные изменения на уровне Fiber Tree, а не через прямое сравнение Virtual DOM.

### 3. Применение изменений:

- После вычисления изменений на уровне Fiber Tree React переходит к фазе “Commit”.
- На этом этапе изменения применяются к реальному DOM: обновляются только те элементы, которые изменились.
- Также запускаются побочные эффекты, такие как useEffect или componentDidUpdate.

# Fiber tree

– Как работает

# Fiber tree

— это не просто статическая структура, а динамическая и гибкая система, которая:

1. Связывает виртуальный DOM с реальным DOM.
2. Хранит состояние приложения (включая хуки и эффекты).
3. Позволяет React эффективно управлять рендерингом:
  - Отслеживает изменения в компонентах.
  - Управляет задачами с разными приоритетами.
  - Оптимизирует перерендеринг через инкрементальный подход.

## Fiber Tree — это объект?

Fiber — это объект, который React создаёт для каждого элемента виртуального DOM. Этот объект содержит всю необходимую информацию для описания элемента и управления его жизненным циклом.

**Каждый объект Fiber содержит множество свойств. Вот основные из них:**

### 1. Тип элемента:

- Указывает, какой тип компонента связан с данным Fiber:
- Стока ('div', 'span' и т. д.) для HTML-элементов.
- Функция или класс для компонентов React.
- null для фрагментов (React.Fragment).

### 2. Ключ (key):

- Используется для идентификации узлов в списках и помогает React отслеживать изменения.

### 3. Пропсы (props):

- Хранят свойства, переданные элементу или компоненту.

### 4. Состояние (state):

- Содержит внутреннее состояние компонента (если это stateful-компонент).

### 5. Родитель, дочерние и соседние узлы:

- return: Ссылка на родительский Fiber.
- child: Ссылка на первый дочерний Fiber.
- sibling: Ссылка на следующий узел на том же уровне.

### 6. Effect List:

- Список побочных эффектов (например, вызов хуков useEffect, изменения в DOM), связанных с этим Fiber.

### 7. Ссылка на DOM-узел:

- Содержит ссылку на соответствующий узел реального DOM (например, <div>), если он уже создан.

### 8. Приоритет обновления:

- Используется для управления порядком выполнения задач (например, обновления с высоким приоритетом для анимаций).

### 9. Алгоритм жизненного цикла:

- Информация о текущем состоянии компонента: например, требуется ли rerender, есть ли эффекты и т. д.

# Fiber tree

— ЭТО ОСНОВА НОВОЙ архитектуры React, представленной в React 16 (2017г) с помощью которой происходит рендеринг компонентов.

## Как работало раньше (до Fiber):

- До появления Fiber React работал синхронно. Когда происходило изменение состояния или пропсов, React полностью обновлял виртуальный DOM и вычислял все изменения, которые нужно внести в реальный DOM. Этот процесс не мог быть прерван.
  - Если, например, рендеринг занимал 5 секунд, всё приложение было заблокировано на это время. Пользовательские взаимодействия, такие как клики, нажатия клавиш или прокрутка, не могли быть обработаны до завершения рендеринга.

## Как работает теперь с Fiber:

- В новой архитектуре Fiber React может задавать приоритеты разным задачам. Например, обработка пользовательских взаимодействий (клики, ввод текста) имеет более высокий приоритет, чем рендеринг менее важных частей интерфейса.
- Если рендеринг занимает много времени, и в это время происходит что-то более важное (например, пользователь кликает по кнопке), React может приостановить текущий рендеринг, выполнить задачу с более высоким приоритетом (например, обработать клик), а затем вернуться к незавершенному рендерингу.
- React может разбивать рендеринг на маленькие части, выполнять их постепенно и между ними проверять, нет ли более приоритетных задач.

## Пример:

### 1. До Fiber:

Пользователь кликает по кнопке, что вызывает обновление большого списка элементов. Рендеринг этого списка занимает 5 секунд. В течение этих 5 секунд интерфейс не отвечает на другие действия. Если пользователь попытается нажать на другую кнопку, она не сработает до завершения рендеринга списка.

### 2. С Fiber:

Пользователь кликает по той же кнопке, и рендеринг списка начинается. Через 2 секунды пользователь кликает по другой кнопке. Благодаря Fiber React приостанавливает рендеринг списка, обрабатывает клик по второй кнопке (более приоритетное действие), и только после этого возвращается к завершению рендеринга списка.

# Fiber tree **vs** Виртуальный ДОМ – В чем разница

# Аналогия из жизни

## Ремонт в доме

Представь, что ты делаешь ремонт в квартире.

Дом как интерфейс:

Твой реальный дом (физическая квартира) – это реальный DOM.

Ты хочешь сделать изменения в доме: перекрасить стены, переставить мебель или заменить обои.

## Роли Virtual DOM и Fiber Tree:

Virtual DOM – это план изменений:

Ты сначала составляешь план – на бумаге или в компьютерной программе.

Ты рисуешь, как будет выглядеть твоя квартира после ремонта (например, новая мебель, обои, цвет стен).

Это аналог Virtual DOM, который описывает, как интерфейс должен выглядеть.

## 2. Fiber Tree – это менеджер ремонта:

Тебе нужен человек, который будет следить за реализацией твоего плана, оценивать, что уже есть в квартире, и минимизировать количество работы.

Например:

Если мебель уже на месте, её не нужно передвигать.

Если стена уже покрашена в нужный цвет, её не нужно перекрашивать.

Этот “менеджер ремонта” – это Fiber Tree.

Он помогает выбрать, что нужно сделать в первую очередь, что можно отложить, и какие работы не нужно выполнять вовсе.

# Аналогия из жизни

Как все работает вместе:

**1. Создание плана (Virtual DOM):**

- Ты создаёшь новую версию своего плана (например, добавляешь новый диван в гостиную).
- Virtual DOM описывает все элементы и изменения, которые должны быть сделаны в квартире.

**2. Сравнение старого и нового плана (Diffing):**

Ты сравниваешь старую версию плана (что было) с новой (что ты хочешь).

Например:

Ты видишь, что стол на кухне стоит на том же месте — его не нужно трогать.

Но в гостиной нужно убрать старый диван и поставить новый.

**3. Управление процессом (Fiber Tree):**

Fiber Tree, как менеджер, определяет, что именно нужно сделать и в каком порядке:

Сначала убрать старый диван.

Потом поставить новый.

Наконец, перекрасить стены.

Fiber Tree также следит за тем, чтобы ты не переделывал работу дважды и выполнял изменения только там, где они нужны.

**4. Выполнение ремонта (Реальный DOM):**

Менеджер (Fiber Tree) отправляет рабочих, чтобы внести изменения в твою квартиру.

Аналогично, React обновляет реальный DOM только там, где это действительно нужно.

**Основное отличие:**

- Virtual DOM — это просто план (что ты хочешь изменить). Сам по себе он не знает, что уже сделано и что в каком порядке делать.
- Fiber Tree — это менеджер, который сравнивает текущую ситуацию с планом, выбирает приоритеты и следит за тем, чтобы работа выполнялась максимально эффективно.

REACT JS

# Сравнительная таблица

# Схема рендеринга

– Как реакт обновляет приложение

# Схема рендеринга

1

## Старт процесса рендеринга

Инициализация программы, вызов функции `setState` или получение новых пропсов в компоненте сигнализирует React о необходимости обновления интерфейса....

2

## Создание нового виртуального DOM

Компоненты возвращают JSX, которые преобразуются в новое дерево виртуального DOM.

3

## Построение и обновление Fiber Tree

На основе виртуального DOM React создает новый Fiber Tree. Каждый узел в Fiber Tree соответствует узлу в виртуальном DOM и хранит информацию о компоненте, его состоянии, пропсах и ссылках на родительские, дочерние и соседние узлы.

Ядро (Механизм работы) Fiber Tree разбивает рендеринг на мелкие задачи (units of work), что позволяет React выполнять рендеринг асинхронно и прерывать его при необходимости.

4

## Reconciliation (Согласование)

React использует механизм работы Fiber для сравнения нового виртуального DOM с предыдущей версией. Это называется diffing. Так React понимает какие части реального DOM нужно обновить.

Fiber помогает React решать, какие узлы виртуального DOM можно переиспользовать, а какие нужно обновить или удалить.

Этот процесс позволяет React минимизировать изменения в реальном DOM.

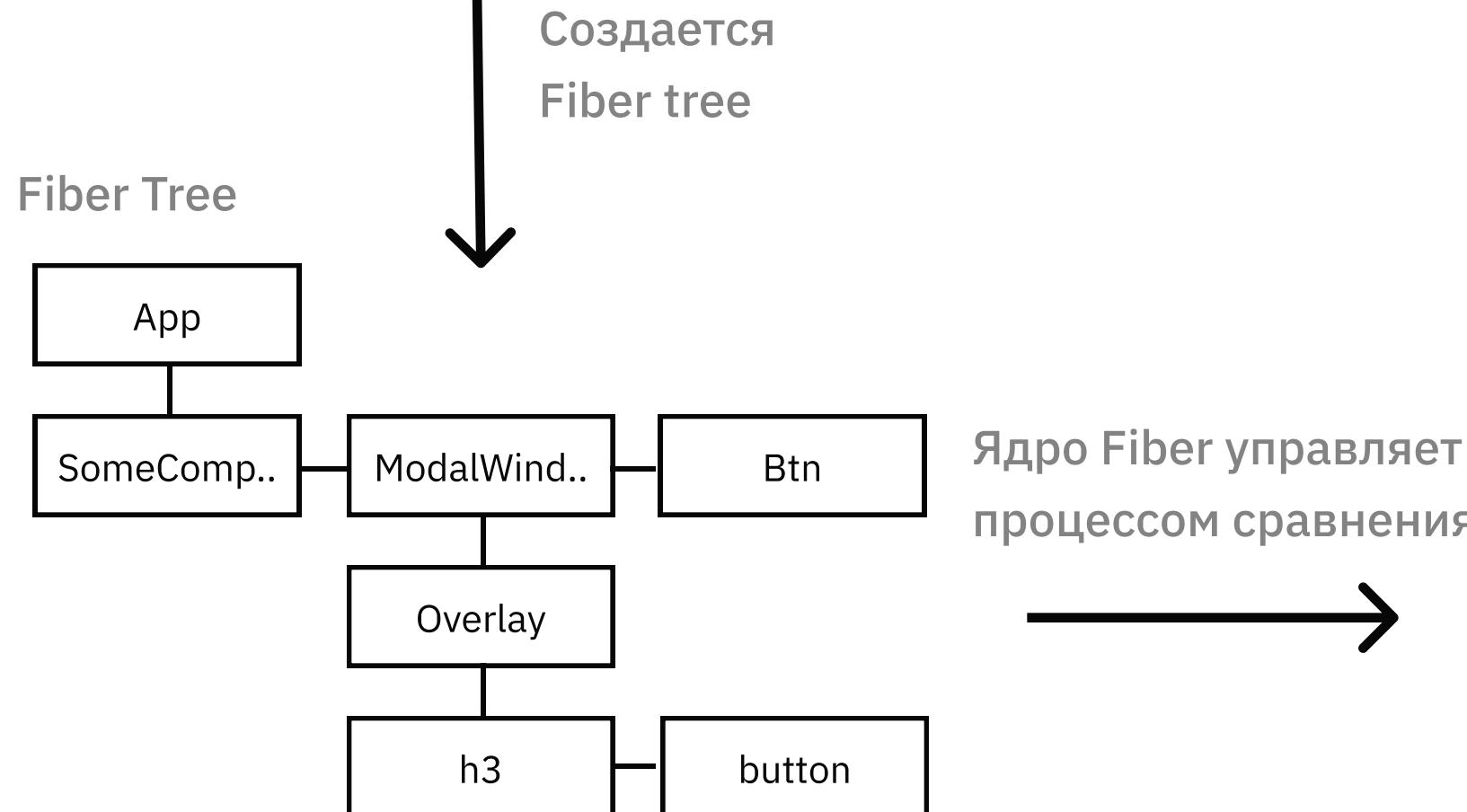
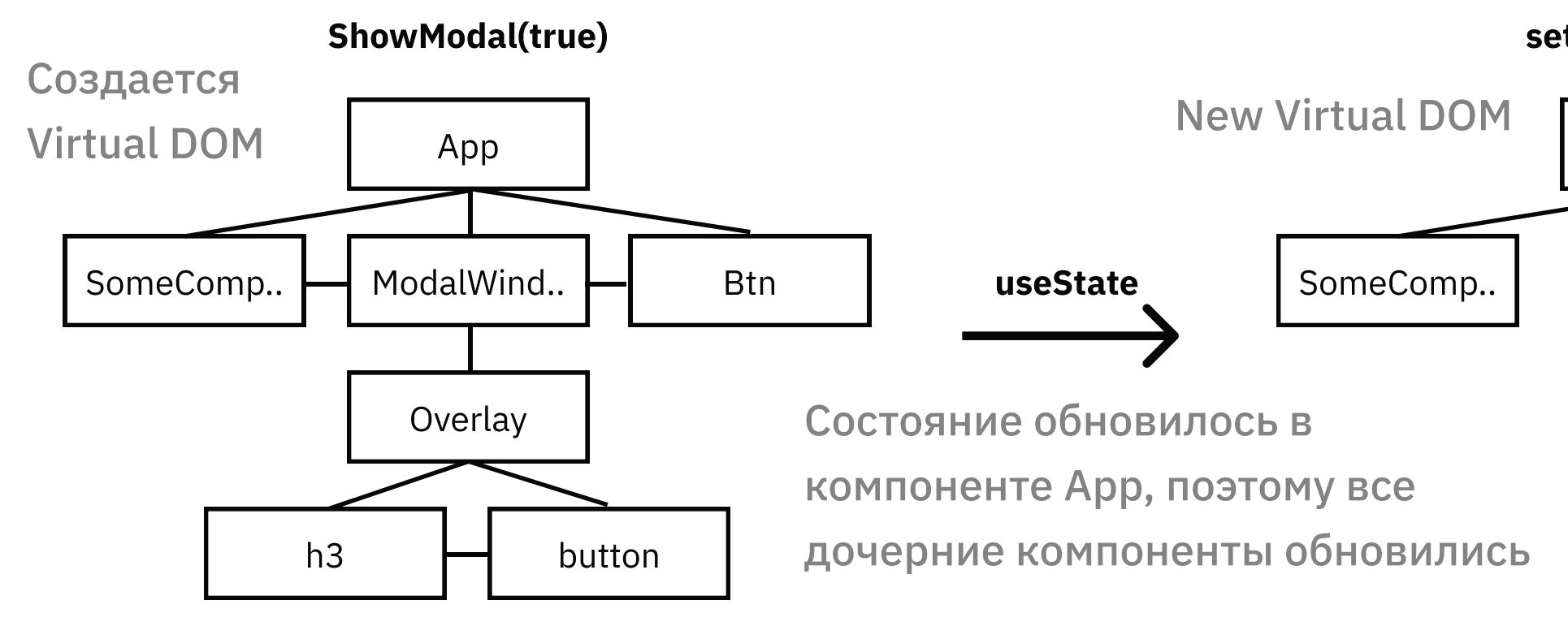
5

## Коммит изменений в реальный DOM (Commit Phase)

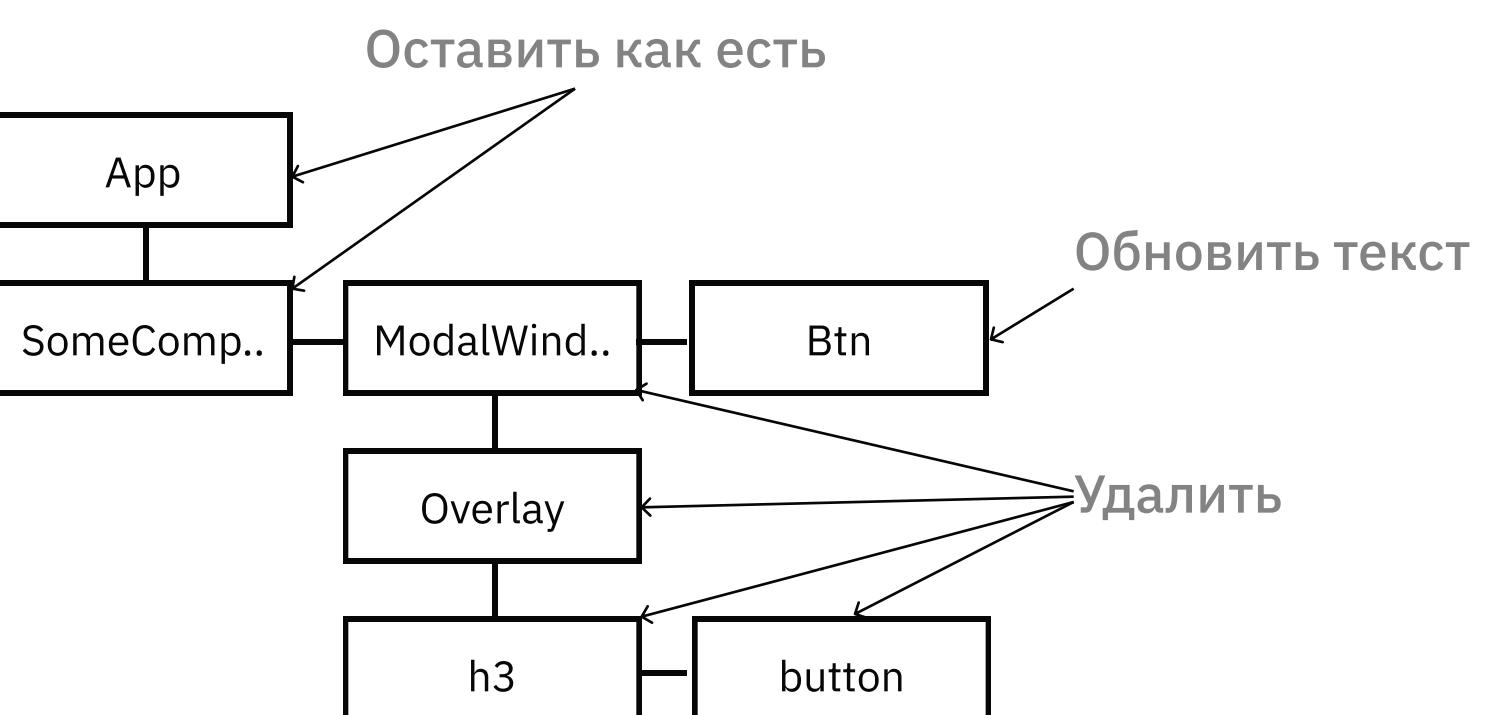
После сравнений React готовит изменения, которые нужно внести в реальный DOM. Этот этап называется commit phase.

React применяет изменения к реальному DOM на основе вычислений. Ядро Fiber контролирует процесс, чтобы убедиться, что только необходимые изменения вносятся в DOM.

# Схема рендеринга на примере

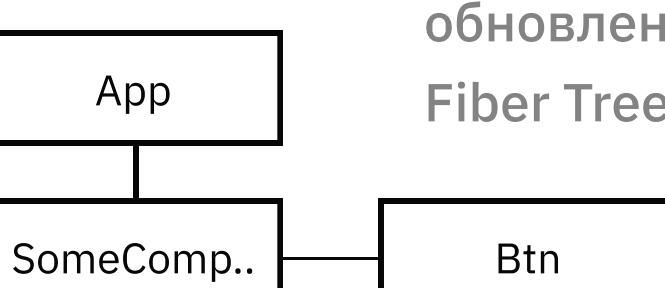


Итоговое решение механизма Fiber об обновлении Fiber Tree



↓

Итоговое обновленное Fiber Tree



Fiber Tree – это не просто копия Virtual DOM, а структура, которая хранит дополнительные данные для оптимизации, например:

- Ссылки на дочерние и родительские узлы.
- Приоритет обновлений (например, для анимаций или пользовательских взаимодействий).
- Хуки и их состояние (например, useState и useEffect).

```

function App() {
  const [showModal, setShowModal] = useState(true);

  return (
    <div className="app">
      <SomeComponent />
      {showModal && (
        <ModalWindow>
          <Overlay>
            <h3>Оставьте отзыв</h3>
            <button>5 звезд!</button>
          </Overlay>
        </ModalWindow>
      )}
      <Btn>{showModal ? "Оценить" : "Закрыть окно"}</Btn>
    </div>
  );
}
  
```

# Мемоизация

– Оптимизация работы приложения

# Ре-рендер

— Ререндеринг дочерних компонентов происходит всегда, если родительский компонент ре-рендерится, **даже если** у дочернего компонента не изменились зависимости, получаемые от родительского компонента (props).

**Такой подход** зачастую расходует **лишние ресурсы** на пере-рисовку и ре-рендеринг

Протестируйте код из комментариев.

1-Увеличьте count - нажатием кнопки

2-Покажите статьи - нажатием кнопки

3-Увеличьте count - нажатием кнопки

Увидите сильную задержку

## На помощь приходит мемоизация

### Мемоизация (Memoization)

Мемоизация — это техника оптимизации, которая заключается в кэшировании результатов вычислений.

Если входные данные не изменяются, повторное вычисление не выполняется — возвращается сохраненный результат.

В React мемоизация используется для предотвращения повторных вычислений и рендеров, если входные данные (зависимости) не изменились.

Это происходит через useMemo, useCallback, и memo.

# memo()

— это функция высшего порядка (Higher-Order Component, HOC), которая предотвращает повторный рендер компонента, если его пропсы не изменились.

— Можете скопировать код из комментариев ниже и протестировать

## Решение проблемы прошлого слайда:

Протестируйте код из комментариев.

- 1-Увеличьте count - нажатием кнопки
- 2-Покажите статьи - нажатием кнопки
- 3-Увеличьте count - нажатием кнопки

Увидите что проблема исчезла

```
const Archive = memo(function Archive({ showPosts }) => {
  // Генерация 30 000 случайных постов
  const [posts] = useState(() => Array.from({length: 30000}, () => createRandomPost()));

  const [showArchive, setShowArchive] = useState(showPosts);

  return (
    <aside>
      <h2>Архив</h2>
      <button onClick={() => setShowArchive(s => !s)}>
        {showArchive ? "Hide archive posts" : "Show archive posts"}
      </button>

      {showArchive && (
        <ul>
          {posts.map((post, i) => (
            <li key={i}>
              <p>
                <strong>{post.title}</strong> {post.body}
              </p>
            </li>
          ))}
        </ul>
      )}
    </aside>
  );
});
```

## memo()

**memo** сравнивает пропсы компонента **Archive** (в данном случае, это **showPosts**). Если пропсы не изменились, **memo** предотвращает рендер компонента. Это помогает улучшить производительность, так как не требуется перерисовывать компонент, если пропсы остаются теми же.

### Когда использовать?

Используйте **memo** для функциональных компонентов, которые перерисовываются слишком часто, но фактически не требуют этого, если пропсы остаются неизменными.

# Фаза коммита

– Commit Phase

# Commit Phase

— заключительный этап  
рендеринга

## Commit Phase (Фаза коммита)

- На этой фазе React применяет все изменения, вычисленные на этапе Render Phase, к реальному DOM.

### Эта фаза включает в себя:

Обновление реального DOM: Применение изменений в структуре DOM, например, добавление, удаление или изменение элементов.

Выполнение эффектов: Для компонентов, использующих useEffect, эффекты, помеченные для выполнения после рендера, также выполняются на этой фазе.

В отличие от Render Phase, которая может быть приостановлена, Commit Phase всегда выполняется синхронно. Это гарантирует, что интерфейс будет полностью обновлен, и все необходимые изменения будут применены в реальном времени.

# ReactDOM

— реальную отрисовку в браузере выполняет не библиотека React, а библиотека ReactDOM

## React & ReactDOM

Реальную отрисовку в браузере выполняет другая библиотека - ReactDOM

```
src > JS index.js > ...
1   import React from "react";
2   import ReactDOM from "react-dom/client";
```

### Такие библиотеки как ReactDOM называют RENDERERS

Хотя мы с вами теперь знаем что настоящий рендер происходит под капотом, а эти библиотеки просто отрисовывают результат.

### Какие еще есть библиотеки RENDERERS для React:

Самый очевидный пример - это React Native

React Native используется для рендеринга мобильных приложений на платформах iOS и Android. Вместо работы с DOM в браузере, React Native преобразует React-компоненты в нативные элементы платформы, такие как View, Text, и Image.

React 360 для создания виртуальных и дополненных реальностей.

Ink позволяет рендерить React-компоненты в консоль или терминал

React PDF используется для генерации PDF-документов с использованием React-компонентов

И многое другое...

# Diffing

– Как работает

# Diffing

— это алгоритм, который React использует для сравнения старого и нового виртуального DOM. Diffing определяет, какие части интерфейса изменились и вычисляет минимальные изменения, которые нужно внести в реальный DOM

## 2 правила по которым работает diffing-сравнение:

Первое: Сравнение по типам элементов/компонентов на одинаковом уровне вложенности.

Если элементы на одном уровне имеют одинаковый тип (например, `<div>` и `<div>`), React считает, что это тот же самый элемент, и сравнивает их атрибуты и содержимое. Если типы совпадают, React делает `shallow compare` (поверхностное сравнение) атрибутов и обновляет только те, которые изменились.

Если типы узлов различаются (например, `<div>` и `<span>`), React удаляет старый элемент и создает новый, поскольку считает, что эти узлы представляют собой разные элементы.

Второе: Сравнение списков (keyed diffing).

Если React сталкивается со списком элементов, он использует специальные ключи (`key`), чтобы отслеживать и сравнивать элементы. Это помогает React точно определять, какие элементы были добавлены, удалены или перемещены.

Ключи должны быть уникальными среди братьев и сестер (`sibling elements`), чтобы React мог правильно отслеживать изменения.

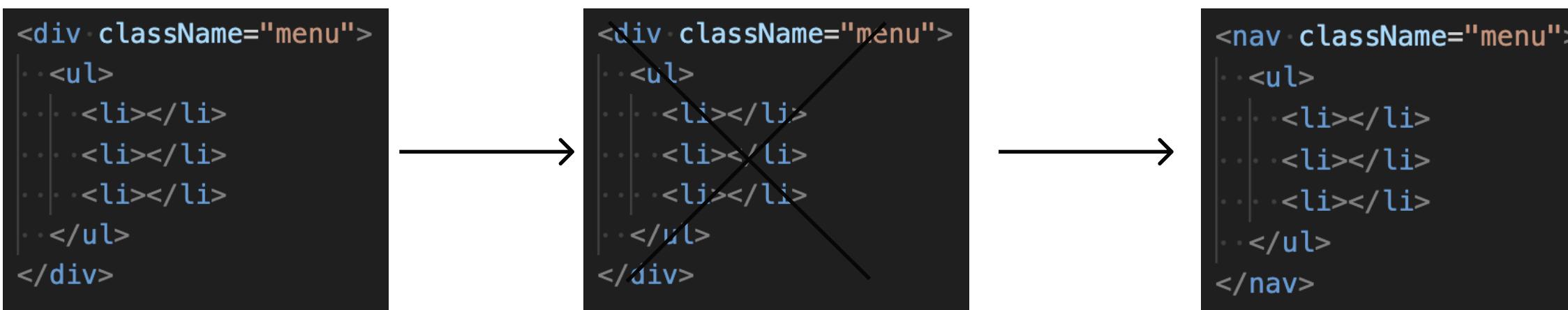
Если React обнаруживает, что текст внутри узла изменился, он обновляет текстовой контент в реальном DOM, не трогая сам элемент.

# Diffing

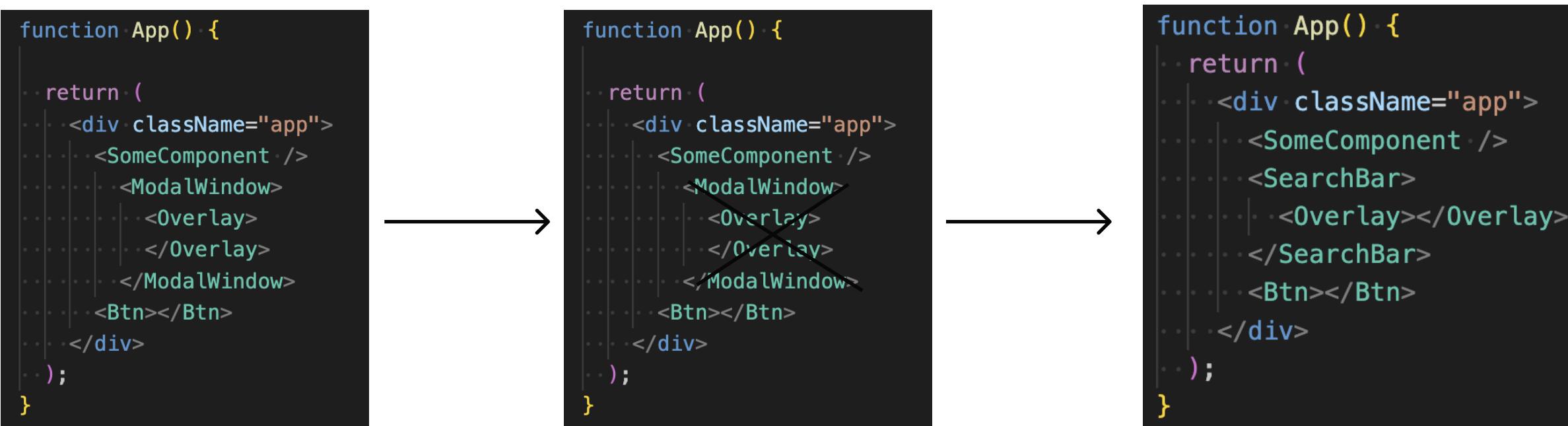
— схема

## Одна и та же позиция, но разные элементы

Значит нужно пересоздать сам элемент и его дочерние элементы и сбросить все state этих элементов



То же самое, если будет компонент, вместо html элемента



# Diffing

— сравнение ключей

— Можете скопировать код из  
комментариев ниже и протестировать

## Одна и та же позиция, одни и те же элементы но без key

Если в нашей программе при изменении state будут создаваться одинаковые элементы в одинаковых местах, это может привести к проблемам при рендеринге этих одинаковых элементов. State будет одинаковым у фактически разных элементов (которые фактически называются одинаково и находятся на одном и том же месте).

*(Напоминаем, что изменение state или props запускает процесс рендеринга, но в процессе рендеринга используется механизм diffing, который уже и принимает решение о том, какую часть программы обновить, а какую оставить без изменений для максимальной эффективности).*

### Пример на табах:

```
function Test() {
  const [activeTab, setActiveTab] = useState(0);
  const tabs = [
    { id: 1, title: "Tab 1" },
    { id: 2, title: "Tab 2" },
    { id: 3, title: "Tab 3" },
  ];
  return (
    <div>
      <" .>
      <div style={{ display: "flex", gap: "10px" }}>
        <" .>
        {tabs.map((tab, index) => (
          <button
            style={{ display: "flex", gap: "10px" }}
            key={tab.id}
            onClick={() => setActiveTab(index)}
          >
            <" .>
            {tab.title}<" .>
            </button>
          ))}<" .>
      </div><" .>
      <div>
        <" .>
        <TabContent key={tabs[activeTab].id} tabId={tabs[activeTab].id}><" .>
      </div><" .>
    </div>
  );
}

function TabContent({ tabId }) {
  const [count, setCount] = useState(0);
  return (
    <div style={{ paddingTop: "32px" }}>
      <" .>
      <h5>Tab {tabId}</h5> <p>Count: {count}</p><" .>
      <button onClick={() => setCount(count + 1)}>Increment</button><" .>
    </div>
  );
}
```

1. Внутри компонента `Test` есть код для кнопок.  
Эти кнопки отображают тот или иной таб.

4-To самое уникальное Key которое помогает Reactу понять что это не один и тот же элемент, а 3 разных.  
(Уберите его и увеличьте инкремент, увидите что у каждого таба одинаковое число)

3-Это создание таба исходя из того какая кнопка нажата. (По идее должно быть 3 кнопки, и 3 таба для каждой кнопки.)

2-Это компонент самого таба.  
В нем есть кнопка, которая увеличивает переменную состояния count на 1.  
Переменная count отображается в контенте таба на странице. Изменение этого состояния запускает рендеринг.

1-Нажимаем на увеличение числа в табе.

2-Запускается процесс рендеринга.

3-Diffing сравнивает новый виртуальный DOM со старым по двум ключевым вещам:

- изменились ли типы элементов и их расположение;
- по ключам;

Так же проверяет текст:

- изменился ли текст внутри элемента? Если да то обновляет только его.

4-Если в компоненте `` нет key, то diffing не обновляет на странице этот элемент, так как он такой же и на том же месте. React только обновляет текст, но state остается.

Если у элемента нет key и если элемент называется одинаково и находится на том же месте что и был, его состояние сохранится при ре-рендеринге.

# Batching updates of states

– Пакетирование обновлений состояний

## Batching updates of states (или “Пакетирование обновлений состояний”)

— это процесс, при котором React объединяет несколько вызовов `setState` в один рендер, чтобы минимизировать количество перерисовок компонента и повысить производительность приложения.

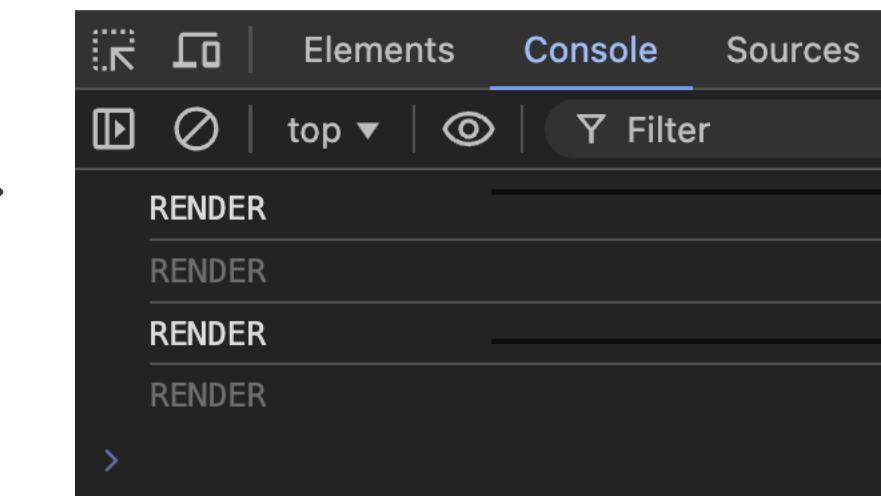
В React 18 пакетирование обновлений стало более умным: теперь React автоматически пакетирует все обновления состояний, происходящие в одном событии или асинхронном коде, таким образом избегая лишних перерисовок.

Batching  
— updates of states

```
export default function App() {
  console.log("RENDER");
  const [number, setNumber] = useState(0);

  function increaseNumByThree() {
    setNumber(number + 1);
    setNumber(number + 1);
    setNumber(number + 1);
  }

  return (
    <div>
      <p style={{fontSize: "24px"}}>{number}</p>
      <button style={{padding: "8px"}} onClick={increaseNumByThree}>
        Увеличь на 3
      </button>
    </div>
  );
}
```



→ Рендер при запуске приложения

→ Рендер после нажатия кнопки

Три обновления состояния должны вызвать три рендера. /  
Но если мы запустим этот код, то в консоли мы увидим слово  
“рендер” только один раз. (Рендер всегда начинается с запуска  
функции компонента.)

# Асинхронное — обновление state из за пакетирования обновлений

## Обновление state происходят асинхронно.

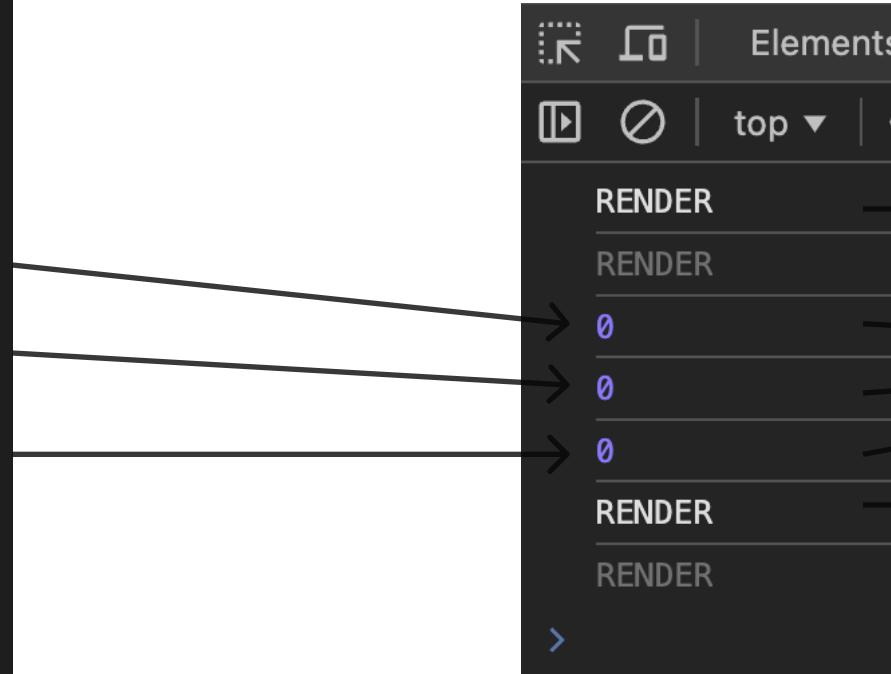
**Пакетирование обновлений (Batching):** React объединяет несколько вызовов `setState` в одно обновление, чтобы избежать избыточного рендеринга и повысить производительность. Это означает, что вызовы `setState` не приводят к немедленному обновлению компонента; вместо этого они откладываются и выполняются одновременно в одном рендере.

```
export default function App() {
  console.log("RENDER");
  const [number, setNumber] = useState(0);

  function increaseNumByThree() {
    setNumber(number + 1);
    console.log(number); // Предполагаем что должно быть 1. Но консоль даст 0
    setNumber(number + 1);
    console.log(number); // Предполагаем что должно быть 2. Но консоль даст 0
    setNumber(number + 1);
    console.log(number); // Предполагаем что должно быть 3. Но консоль даст 0
  }

  return (
    <div>
      <p style={{ fontSize: "24px" }}>{number}</p>
      <button style={{ padding: "8px" }} onClick={increaseNumByThree}>
        Увеличь на 3
      </button>
    </div>
  );
}
```

**Вопрос!?**  
Что в итоге выведется на экран после нажатия  
кнопки?  
Число 3 или 1?



Три обновления состояния должны вызвать три рендера. /  
Но если мы запустим этот код, то в консоли мы увидим слово  
“рендер” только один раз. (Рендер всегда начинается с запуска  
функции компонента.)

**Ответ:**  
Число 1

Из за асинхронности обновления состояний,  
которое сделано для оптимизации, реакт после  
нажатия кнопки будет делать следующее:

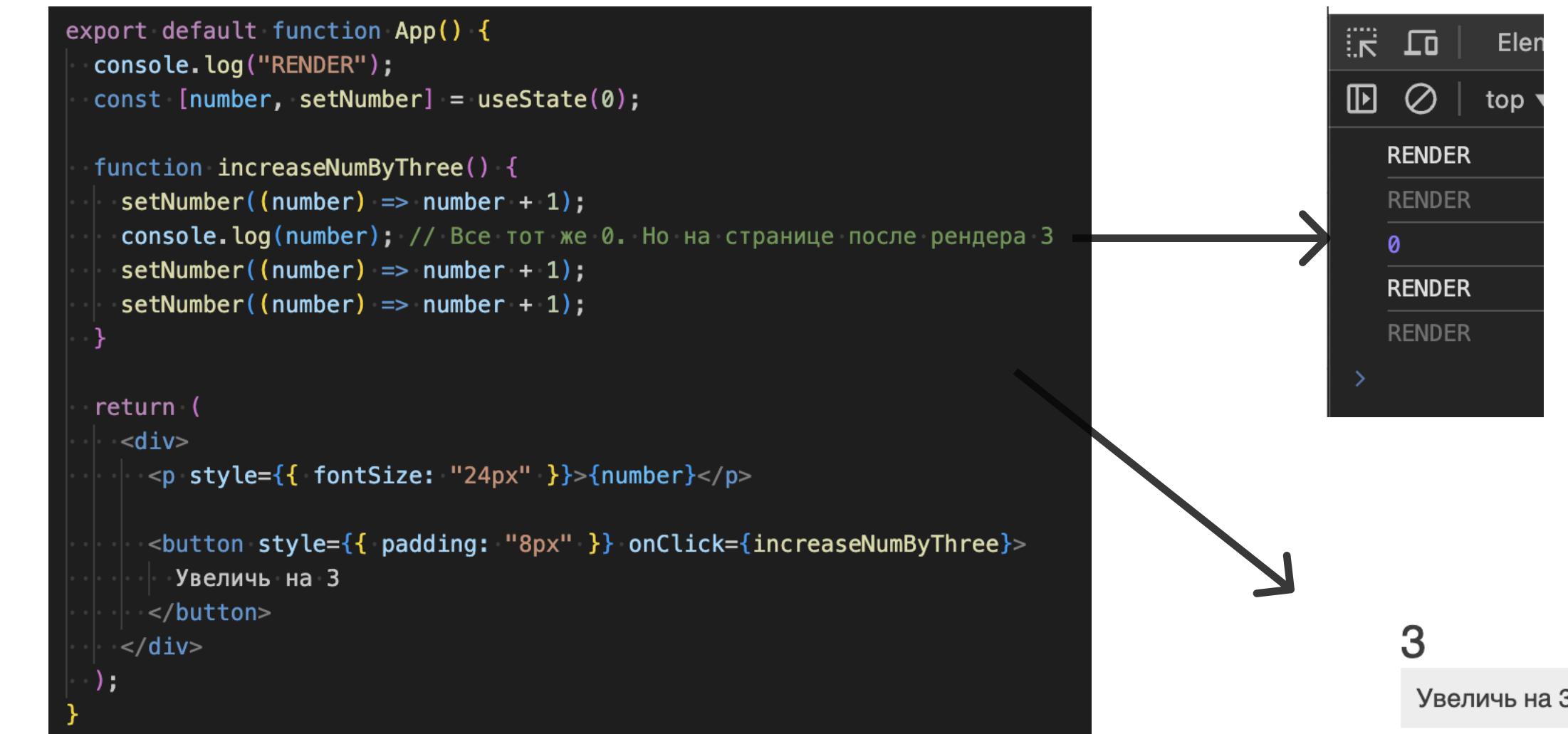
- 1.Запустится функцию `increaseNumByThree()`
- 2.Реакт соберет в Banch все три обновления состояния и обновит их по очереди,  
но обновлять он их будет от нуля (старого состояния) все три раза
- 3.Запустит 1 рендеринг и отобразит результат

# Использование обновленного значения

— как обновить текущее значение и  
использовать его при необходимости

## Правило!

Когда нужно обновить состояние на основе его старого состояния,  
старое состояние нужно получить из call-back функции setState  
`setState((prevState) => prevState + 1);`



```
export default function App() {
  console.log("RENDER");
  const [number, setNumber] = useState(0);

  function increaseNumByThree() {
    setNumber((number) => number + 1);
    console.log(number); // Все тот же 0. Но на странице после рендера 3
    setNumber((number) => number + 1);
    setNumber((number) => number + 1);
  }

  return (
    <div>
      <p style={{ fontSize: "24px" }}>{number}</p>
      <button style={{ padding: "8px" }} onClick={increaseNumByThree}>
        Увеличь на 3
      </button>
    </div>
  );
}
```

# Mount & Unmount

– Компонентов

# Mount & Unmount

— Монтирование и размонтирование  
компонентов

В React компоненты проходят несколько жизненных циклов, и важными этапами этого цикла являются монтирование (mount) и размонтирование (unmount)

## 1 Монтирование (Mount)

- Компонент рендерится в первый раз
- Создаются состояния и props

## 2 Re-render (Опционально) происходит когда:

- Изменяются State
- Изменяются props
- Ре-рендерится родительский компонент
- Меняется контекст. (Об этом позже)

Ре-рендер может происходить  
множество раз

## 2 Размонтирование (Unmount)

- Компонент удаляется
- Состояния и props уделяются

# Framework vs library

— И ИХ ОТЛИЧИЯ

# Library

— Контролируешь сам каких специалистов использовать  
(Какие пакеты использовать для решения тех или иных задач)

## Аналогия: Ремонт дома

### Библиотека: Вы сами нанимаете специалистов для конкретных задач

Представь, что ты делаешь ремонт в своей квартире, и тебе нужны разные специалисты: электрик, плиточник, маляр и т. д.

Ты сам принимаешь решения, кого и когда вызывать, и каждый специалист выполняет только свою узкую задачу (например, плиточник кладёт плитку, а электрик проводит проводку).

В этом случае ты контролируешь весь процесс ремонта, а специалисты просто помогают выполнять конкретные задачи.

### Пример из программирования:

Библиотеки (например, React, lodash, jQuery) — это набор готовых инструментов, которые ты вызываешь, когда они тебе нужны.

### Фреймворк: Ты нанимаешь компанию для “ремонта под ключ”

Представь, что ты не хочешь управлять ремонтом сам, а нанимаешь строительную компанию.

Она берет на себя весь процесс: планирование, закупку материалов, координацию работы специалистов и сдачу готового результата.

Компания диктует правила работы: ты можешь влиять только на некоторые детали (например, выбрать цвет стен), но не управляешь, как и в какой последовательности всё делается.

В этом случае фреймворк управляет процессом, а ты лишь предоставляешь необходимые данные.

### Пример из программирования:

- Фреймворки (например, Angular, Django) контролируют весь процесс разработки.

Ты следуешь их структуре и предоставляешь только определённые части логики.

# useRef

– сохраняет данные между рендерами

# useRef()

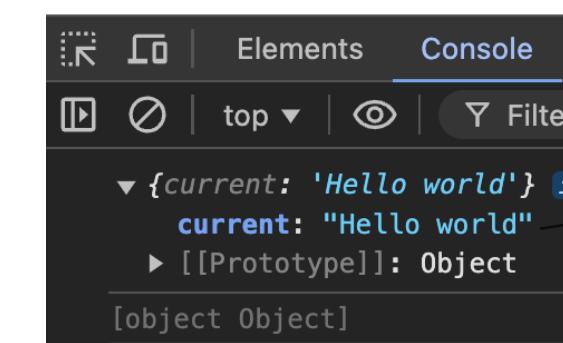
— часто используется для работы с DOM-элементами, хранения значений, которые остаются неизменными между рендерами, и для создания переменных, которые не должны инициировать повторный рендеринг компонента.

## useRef() что это,

useRef() - это хук для создания переменной которая будет объектом.

Значение которое передается в useRef(значение) хранится в свойстве current

```
1 import {useRef} from "react";
2
3 export default function App() {
4
5   const refVariable = useRef("Hello world");
```



Объект с свойством current

useRef() - сохраняет свое значение между рендерами.  
Но не запускает Ре-рендер.

## 3 основных случая использования useRef()

### 1. Хранение DOM-элементов:

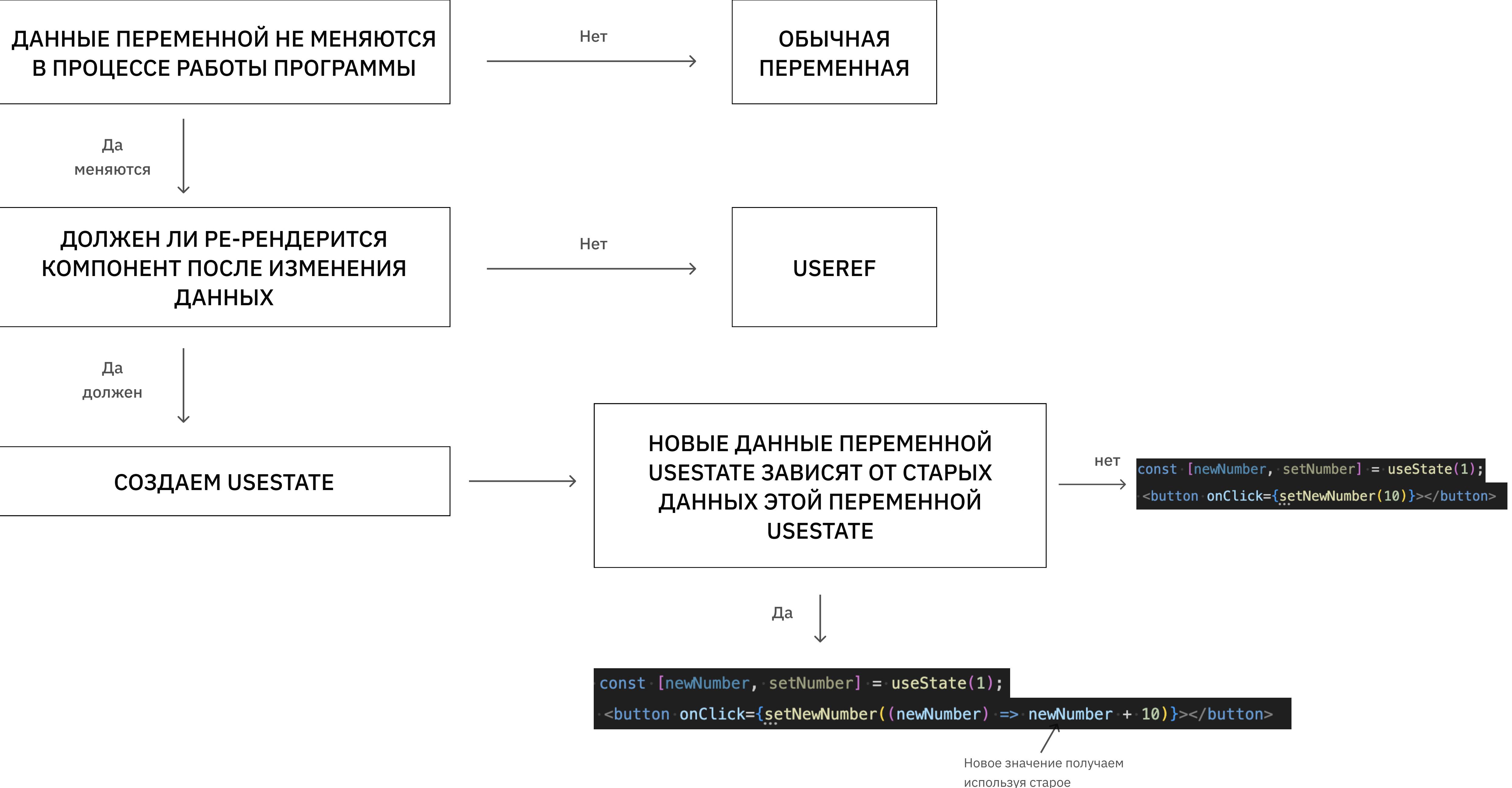
- useRef часто используется для доступа к DOM-элементам напрямую. Это полезно, когда нужно управлять элементом вручную, например, сфокусироваться на инпуте.

### 2. Хранение изменяемых значений:

- Вы можете использовать useRef для хранения значений, которые нужно изменять без повторного рендеринга компонента. В отличие от useState, изменение значения useRef не вызывает перерисовку компонента.

### 3. Сохранение значений между рендерами:

- Значение, сохранённое в useRef, остаётся неизменным между рендерами, что делает его идеальным для хранения значений, которые должны “пережить” обновления компонента.



# useRef()

— для хранения DOM элемента

## useRef() для хранения DOM элемента

1. Создаём компонент.
2. Создаём переменную и присваиваем ей значение useRef(null).
3. Компонент возвращает HTML-элемент.
4. В атрибутах HTML-элемента прописываем prop ref со значением переменной, созданной с помощью хука useRef(): ref={refElement}.

*Таким образом, в переменной, созданной с помощью хука useRef(), всегда будет храниться ссылка на этот элемент.*

```
function Input() {  
  const refElement = useRef(null);  
  
  useEffect(() => {  
    refElement.current.focus();  
  }, []);  
  
  return <input ref={refElement}>;  
}
```

**Например:**

Для того чтобы элемент <input> автоматически фокусировался после загрузки страницы, мы используем хук useEffect, который запускается один раз после рендера.

(Используем useEffect(), так как нам нужно сначала создать всю страницу и элементы, и только затем что-то сделать с этими элементами).

# кастомный Хук

— сделай сам

# Кастомные хуки

— это функции, которые позволяют повторно использовать логику состояния и эффекты в нескольких компонентах

## Для чего нужны кастомные хуки?

### 1. Повторное использование логики:

- Если вы замечаете, что одна и та же логика используется в нескольких компонентах, вы можете вынести её в кастомный хук, чтобы избежать дублирования кода.

### 2. Организация кода:

- Кастомные хуки помогают разделить сложные компоненты на более мелкие, легко управляемые части, улучшая читаемость и поддерживаемость кода.

### 3. Инкапсуляция логики:

- Кастомные хуки позволяют инкапсулировать логику, скрывая её от основного компонента, что делает компонент более простым и понятным.

## 2 правила создания кастомных Хуков

1- Кастомных хук должен использовать хотя бы 1 стандартный хук (useState, useEffect, useRef, useContext, и так далее)

2- Имя функции кастомного хука начинается с префикса “use”

# Пример #1

— Кастомный хук который выводит получает данные из API

## Кастомный Хук

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch(url);
        if (!response.ok) {
          throw new Error(`HTTP Error: ${response.status}`);
        }
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    }

    fetchData();
  }, [url]);

  return { data, loading, error };
}

export default useFetch;
```

## Использование хука в компоненте

```
import useFetch from "./hooks/useFetch";

function Posts() {
  const { data, loading, error } = useFetch("https://jsonplaceholder.typicode.com/posts");

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <ul>
      {data.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default Posts;
```

## Пример #2

- Кастомный хук который выводит ширину и высоту окна браузера

```
import { useState, useEffect } from 'react';

// Кастомный хук
function useWindowSize() {
  const [windowSize, setWindowSize] = useState({
    width: window.innerWidth,
    height: window.innerHeight
  });

  useEffect(() => {
    // Обработчик события изменения размера окна
    function handleResize() {
      setWindowSize({
        width: window.innerWidth,
        height: window.innerHeight
      });
    }

    // Добавляем обработчик события
    window.addEventListener('resize', handleResize);

    // Убираем обработчик события при размонтировании компонента
    return () => window.removeEventListener('resize', handleResize);
  }, []); // Пустой массив зависимостей указывает на выполнение эффекта только при монтировании

  return windowSize; // Возвращаем текущее значение размера окна
}

// Использование кастомного хука в компоненте
export default function MyComponent() {
  const windowSize = useWindowSize();

  return (
    <div>
      <h1>Размер окна</h1>
      <p>Ширина: {windowSize.width}px</p>
      <p>Высота: {windowSize.height}px</p>
    </div>
  );
}
```

## Пример #2

— Кастомный хук с параметрами

Кастомный хук может принимать аргументы как обычная функция

```
import { useState, useEffect } from 'react';

function useWindowSize(minWidth, maxWidth) {
  const [isWithinRange, setIsWithinRange] = useState(false);

  useEffect(() => {
    function handleResize() {
      const width = window.innerWidth;
      setIsWithinRange(width >= minWidth && width <= maxWidth);
    }

    handleResize(); // Проверка при первой загрузке
    window.addEventListener('resize', handleResize);

    return () => window.removeEventListener('resize', handleResize);
  }, [minWidth, maxWidth]);

  return isWithinRange;
}

// Использование кастомного хука в компоненте
export default function MyComponent() {
  const isWithinRange = useWindowSize(500, 1200);

  return (
    <div>
      <h1>Размер окна</h1>
      {isWithinRange ? (
        <p>Ширина окна в пределах заданного диапазона.</p>
      ) : (
        <p>Ширина окна вне заданного диапазона.</p>
      )}
    </div>
  );
}
```

— Можете скопировать код из комментариев ниже и протестировать

# React Router

– библиотека

# React Router 6.4+

— Это библиотека для управления маршрутизацией в приложениях на базе React.

Она позволяет разработчикам создавать одностраничные приложения (SPA) с несколькими “страницами” или разделами, сохраняя при этом плавность и быстроту работы без полной перезагрузки страницы

## Современный синтаксис

## Установка библиотеки в проект:

В папке с проектом, в терминале прописать:

npm i react-router-dom

```

    <pre>
import { createBrowserRouter, RouterProvider } from "react-router-dom";
import Product from "./pages/Product";
import Pricing from "./pages/Pricing";
import Homepage from "./pages/Homepage";
import PageNotFound from "./pages/PageNotFound";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Homepage />, // Главная страница
  },
  {
    path: "/product",
    element: <Product />, // Страница продукта
  },
  {
    path: "/pricing",
    element: <Pricing />, // Страница с ценами
  },
  {
    path: "*",
    element: <PageNotFound />, // 404
  },
]);
function App() {
  return <RouterProvider router={router} />;
}

export default App;
    </pre>

```

The diagram illustrates the setup of a React Router application. On the left, a file tree shows the directory structure: a 'pages' folder containing 'Homepage.jsx', 'PageNotFound.jsx', 'Pricing.jsx', 'Product.jsx', 'App.jsx', and 'main.jsx'. Lines connect each of these files to specific parts of the code on the right. The code defines a 'createBrowserRouter' function with four routes: one for the homepage ('/'), one for the product page ('/product'), one for the pricing page ('/pricing'), and a catch-all route ('\*') for a 404 page. It also defines an 'App' component that wraps the router provider. Below the code, three browser screenshots show the results of navigating to '/misstake', '/pricing', and '/product' respectively. The first screenshot shows a 'Not found' message, the second shows the 'Pricing' page, and the third shows the 'Product' page.

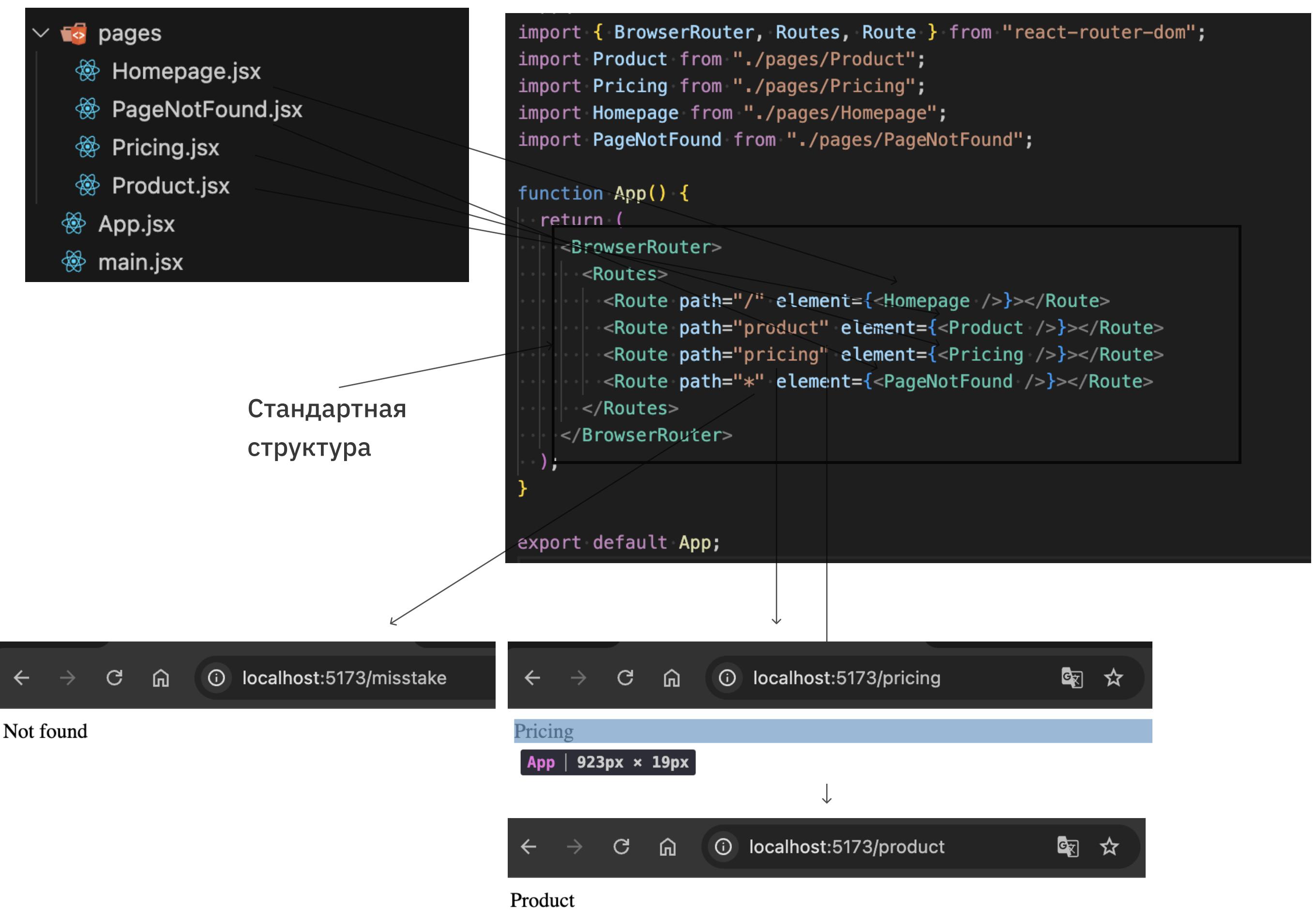
# React Router до 6.4

Устаревший синтаксис. До версии 6.4

## Установка библиотеки в проект:

В папке с проектом, в терминале прописать:

npm i react-router-dom



# Router <Link>

— как работают ссылки

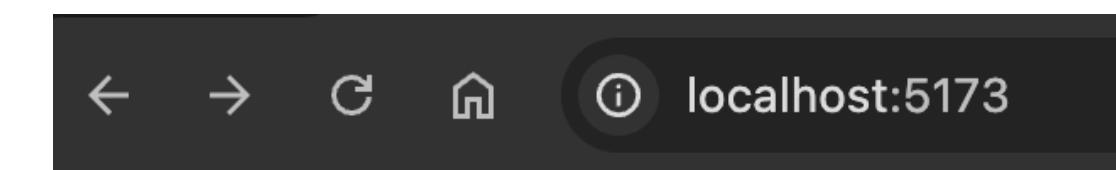
## Ссылки <Link>

— В React Router работают как навигационные элементы, которые позволяют пользователям перемещаться между различными “страницами” или разделами вашего приложения, не перезагружая страницу.

## Как работают ссылки в router:

Компонент `<Link>` в React Router используется для создания ссылок внутри приложения. Атрибут `to` указывает на путь, к которому нужно перейти.

Он может быть строкой (например, `/about`) или объектом, который позволяет более гибко настраивать навигацию.



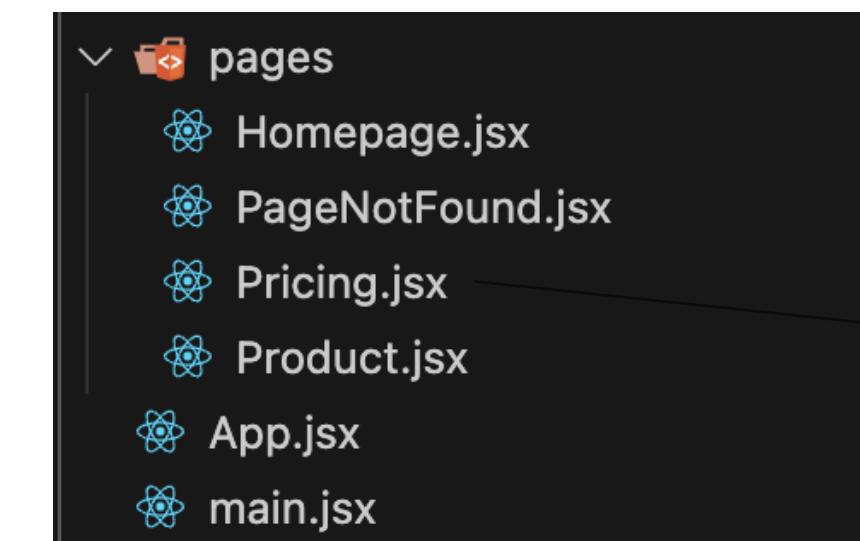
### Home

[Pricing](#)

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

function Homepage() {
  return (
    <div>
      <h1>Home</h1>
      <Link to="/pricing">Pricing</Link>
    </div>
  );
}

export default Homepage;
```



```
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Homepage />}></Route>
        <Route path="product" element={<Product />}></Route>
        <Route path="pricing" element={<Pricing />}></Route>
        <Route path="*" element={<PageNotFound />}></Route>
      </Routes>
    </BrowserRouter>
  );
}
```

## Ссылки <Link>

— Дополнительные возможности.

Объект вместо текста

**Объект может включать дополнительные свойства, такие как:**

- **pathname** — путь к странице.
- **search** — параметры строки запроса (например, ?sort=asc).
- **hash** — якорь на странице (например, #section1).
- **state** — данные, которые можно передать в маршрут.

```
<Link
  to={{
    pathname: "/about",
    search: "?sort=asc",
    hash: "#info",
    state: { fromHome: true }
  }}
>
  About
</Link>
```

- **pathname: "/about"** — указывает путь к странице.
- **search: "?sort=asc"** — добавляет параметры запроса к URL.
- **hash: "#info"** — прокручивает страницу к элементу с идентификатором info.
- **state: { fromHome: true }** — передает данные (например, fromHome), которые можно использовать в целевом компоненте через хук useLocation.

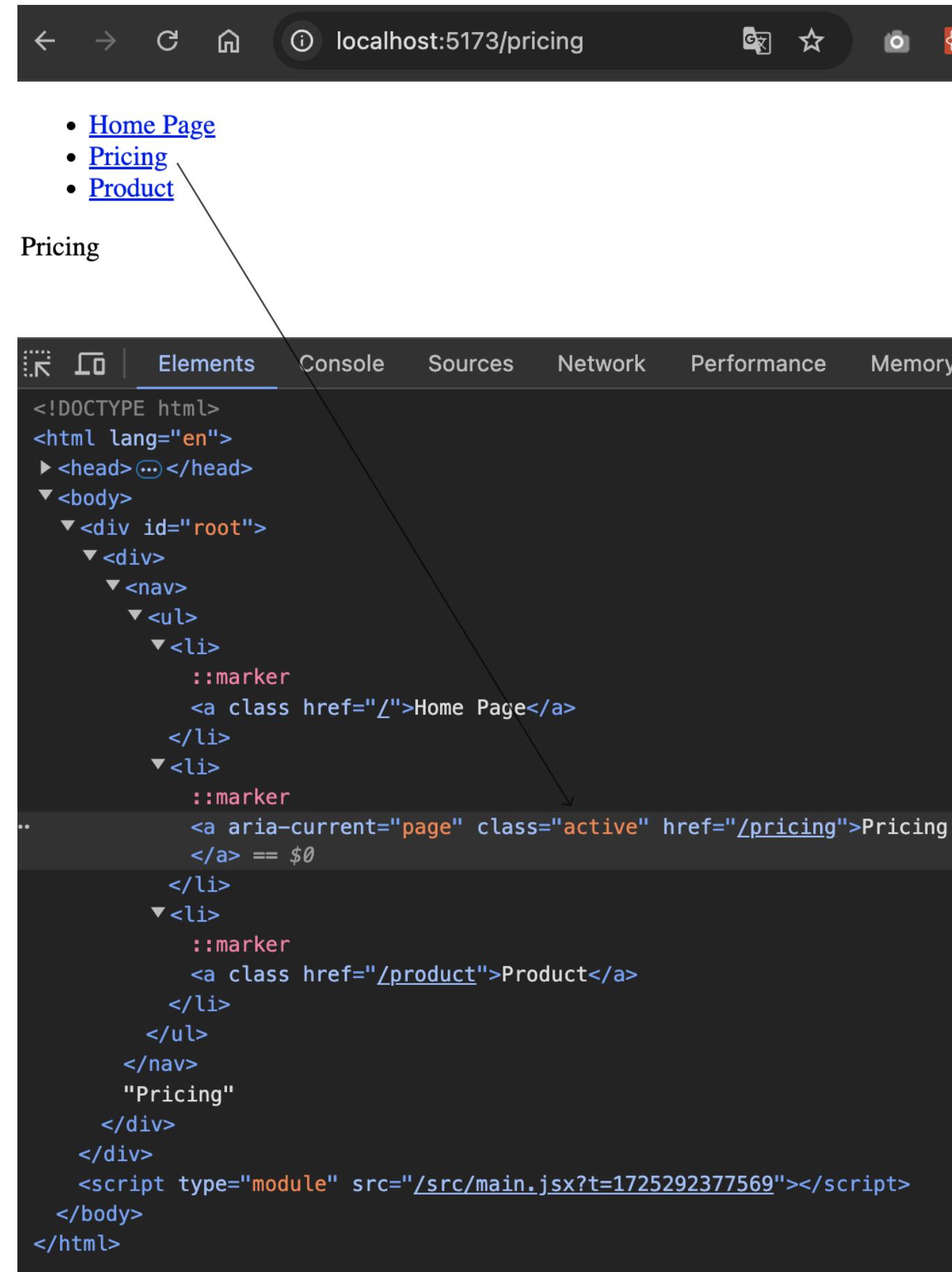
# Ссылки <NavLink>

— Это специальный компонент в React Router, который является расширением <Link>.

Он используется для создания ссылок с активным состоянием (например, выделение текущей активной ссылки).

## Пример <NavLink>:

К нажатой ссылке добавляется класс **active**



```
import { NavLink } from "react-router-dom";

function PageNav() {
  return (
    <nav>
      <ul>
        <li>
          <NavLink to="/">Home Page</NavLink>
        </li>
        <li>
          <NavLink to="/pricing">Pricing</NavLink>
        </li>
        <li>
          <NavLink to="/product">Product</NavLink>
        </li>
      </ul>
    </nav>
  );
}

export default PageNav;
```

# <Outlet />

– как {children} только для страниц

# <Outlet />

— ЭТО КОМПОНЕНТ,  
предоставляемый React Router,  
который используется в  
компоненте маршрута для  
рендеринга дочерних  
маршрутов.  
Он работает как “месторазметка”  
в вашем компоненте, где будут  
отображаться дочерние  
элементы маршрутов.

## Структура маршрутов:

- Когда у вас есть маршруты, которые включают в себя дочерние маршруты (вложенные маршруты), вы можете использовать <Outlet /> в родительском компоненте для указания места, где должны отображаться эти дочерние маршруты.
- Родительский маршрут рендерит свой компонент, и если в этом компоненте есть <Outlet />, то в этом месте будет рендериться соответствующий дочерний маршрут.

```
import {
  createBrowserRouter,
  RouterProvider,
} from "react-router-dom";
import Home from "./Home";
import About from "./About";
import Dashboard from "./Dashboard";
import Settings from "./Settings";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />, // Root route
  },
  {
    path: "about",
    element: <About />, // Route for the About page
  },
  {
    path: "dashboard",
    element: <Dashboard />, // Route for the Dashboard
    children: [
      {
        path: "settings",
        element: <Settings />, // Nested route for Settings
      },
    ],
  },
]);
function App() {
  return <RouterProvider router={router} />;
}
export default App;
```

В этом примере Dashboard является родительским маршрутом, а Settings – дочерним. Для рендеринга <Settings /> внутри <Dashboard />, вы добавляете <Outlet /> в компонент Dashboard:

```
import { Outlet } from 'react-router-dom';

function Dashboard() {
  return (
    <div>
      <h2>Dashboard</h2>
      {/* Здесь будут рендериться дочерние маршруты */}
      <Outlet />
    </div>
  );
}

export default Dashboard;
```

### Работа <Outlet />:

- Когда пользователь переходит на /dashboard, отображается компонент Dashboard.
- Если пользователь переходит на /dashboard/settings, то компонент Settings рендерится внутри компонента Dashboard, именно в том месте, где находится <Outlet />.

Router, атрибут **index**  
— дочерняя страница по умолчанию

# Router, атрибут index

— используется для указания, какой маршрут должен быть отображен по умолчанию, если родительский маршрут совпадает с URL, но не указан конкретный дочерний маршрут

## Как работает index:

### Основная идея:

- Вместо создания отдельного маршрута с конкретным путем, вы создаете маршрут с атрибутом index, который будет рендериться, если путь совпадает с родительским, но не указан конкретный дочерний маршрут.

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Layout />,
    children: [
      { index: true, element: <Home /> },
      { path: "about", element: <About /> },
      { path: "cart", element: <Cart /> },
      { path: "categories", element: <Categories /> },
      { path: "product", element: <ProductDetails /> },
      { path: "*", element: <NotFound /> },
    ],
  },
]);
```

- Родительский маршрут /dashboard рендерит компонент Dashboard.
- Если пользователь переходит на /dashboard, но не указывает какой-либо дочерний маршрут, React Router автоматически рендерит компонент, связанный с маршрутом index, то есть Overview.
- Если пользователь переходит на /dashboard/settings, то рендерится компонент Settings.

# Router `useParams()`

– Хук. Получает **параметр** текущего маршрута

# useParams()

— это хук из библиотеки React Router, который позволяет вам получить параметры текущего маршрута. Он возвращает объект, где ключами являются имена параметров, определенные в вашем маршруте, а значениями — текущие значения этих параметров.

## Как это работает:

### 1. Определение маршрута с параметрами:

Параметры маршрута задаются с помощью двоеточия (:) перед именем параметра в пути маршрута. Здесь :id — это параметр маршрута.

```
{
  path: "/categories/:categoryId",
  element: (
    <>
      <Header />
      <Categories />
    </>
  ),
},
```

### 2. Использование useParams() в компоненте:

Когда маршрут активен (т.е. текущий URL совпадает с маршрутом, определенным в Route), React Router будет извлекать значение параметра из URL и передавать его через useParams().

Пример использования:

```
import { useParams } from "react-router-dom";

function Categories() {
  const id = useParams();
  console.log(id); // {categoryId: '3'}
  return <div>Categories</div>;
}

export default Categories;
```

### 3. Получение параметров:

Если текущий URL, например, /cities/123, то вызов useParams() вернет объект { id: "123" }. Вы можете использовать это значение внутри компонента для получения данных, отображения информации и т.д.

# Router **useSearchParams()**

– Хук получать, изменять и добавлять query string

# useSearchParams()

— ЭТО ХУК, который предоставляет возможность работы с параметрами строки запроса (query parameters) в URL в приложениях React, использующих библиотеку react-router-dom.

С его помощью можно получать, изменять и добавлять параметры поиска в URL без необходимости перезагрузки страницы.

## Что такое параметр запроса (query string):

— это то, что идёт после знака ? в URL. Например: <http://example.com/?name=John>  
Здесь name – это название параметра, а John – его значение.

Вы можете программно обновить URL, добавив или изменив параметры, например, с ?name=John на ?name=Jane. При этом страница не будет перезагружаться, но ваше приложение увидит изменения и может на них отреагировать (например, отфильтровать список или показать другие данные).

## Когда это используется?

Представьте, что у вас есть страница со списком товаров. Вы хотите при клике на фильтр “Показать только распродажу” изменить URL так, чтобы он включал параметр ?sale=true. Тогда, если пользователь сохранит эту ссылку или обновит страницу, фильтр продолжит работать.

## Как работает:

При вызове useSearchParams() вы получаете кортеж из двух значений:

```
const [searchParams, setSearchParams] = useSearchParams();
```

searchParams — объект, с помощью которого мы читаем текущие параметры из адресной строки.  
setSearchParams — функция, чтобы изменить параметры.

### Получение параметра:

`searchParams.get('name')` вернёт значение параметра name. Если параметра нет, будет null.

### Изменение параметров:

- `setSearchParams({ name: value })` установит в URL параметр ?name=..., где ... — это ваше value.

# useSearchParams()

— это хук, который предоставляет возможность работы с параметрами строки запроса (query parameters) в URL в приложениях React, использующих библиотеку react-router-dom.

С его помощью можно получать, изменять и добавлять параметры поиска в URL без необходимости перезагрузки страницы.

## Как это работает:

Предположим, у нас есть URL вида: /user/:id, и в нем также есть строка запроса, например, /user/123?name=Polina&age=25.

```
import React from 'react';
import { useParams, useSearchParams } from 'react-router-dom';

function UserProfile() {
  // Use the useParams hook to get the path parameter `id`
  const { id } = useParams();

  // Use the useSearchParams hook to get and update the query string parameters
  const [searchParams, setSearchParams] = useSearchParams();

  // Get query string parameters
  const name = searchParams.get('name'); // Polina
  const age = searchParams.get('age'); // 25

  // Function to update the query string parameters
  const updateSearchParams = () => {
    // Replace the query string parameters
    setSearchParams({ name: 'Polina', age: 30 });
  };

  return (
    <div>
      <h1>User Profile</h1>
      <p>User ID: {id}</p> /* Path parameter */
      <p>Name: {name}</p> /* Query string parameter */
      <p>Age: {age}</p> /* Query string parameter */
      <button onClick={updateSearchParams}>Update Name and Age</button>
    </div>
  );
}

export default UserProfile;
```

**Вызов setSearchParams приводит к повторному рендерингу компонента, в котором используется хук useSearchParams.**

**Это похоже на то, как вызов setState обновляет состояние и вызывает повторный рендер.**

# Объект searchParams

— Кроме get() вы можете использовать и другие методы, такие как:

## Как это работает:

**has(key):** Проверяет, существует ли параметр с указанным ключом.

```
const hasName = searchParams.has('name'); // true или false
```

**getAll(key):** Возвращает массив всех значений для параметра с данным ключом (если один и тот же параметр может повторяться в строке запроса несколько раз).

```
const allTags = searchParams.getAll('tag'); // Например: ['react', 'router']
```

**keys():** Возвращает итератор по всем ключам параметров.

```
for (const key of searchParams.keys()) {  
  console.log(key);  
}
```

**values():** Возвращает итератор по всем значениям параметров.

```
for (const value of searchParams.values()) {  
  console.log(value);  
}
```

**entries():** Возвращает итератор по всем парам [ключ, значение].

```
for (const [key, value] of searchParams.entries()) {  
  console.log(key, value);  
}
```

**forEach(callback):** Вызывает колбэк-функцию для каждой пары ключ-значение.

```
searchParams.forEach((value, key) => {  
  console.log(` ${key}: ${value}`);  
});
```

# Router **useLocation()**

– Хук. Получает текущий URL

# Router useLocation()

— это хук, предоставляемый React Router, который позволяет получить объект текущего URL.

Этот объект содержит информацию о текущем пути, строке запроса, хэше и состоянии, переданном при навигации.

## Когда использовать useLocation?

**useLocation полезен в следующих случаях:**

1. Когда вам нужно узнать текущий путь (pathname).
2. Для работы с параметрами строки запроса (search query).
3. Для получения состояния, переданного через navigate (например, дополнительной информации о маршруте).
4. Для отслеживания изменения пути в приложении.
5. Для аналитики

```
import { useLocation } from 'react-router-dom';

function CurrentLocation() {
  const location = useLocation();

  console.log(location);

  return (
    <div>
      <h1>Current Location</h1>
      <p>Pathname: {location.pathname}</p>
      <p>Search Query: {location.search}</p>
      <p>Hash: {location.hash}</p>
      <p>State: {JSON.stringify(location.state)}</p>
    </div>
  );
}

export default CurrentLocation;
```

REACT JS

## Router useLocation()

— возвращает объект.

**useLocation возвращает объект:**

```
const location = useLocation();
console.log(location);
```

```
▼ {pathname: '/category/Electronics', search: '?maxPrice=900', hash: '', state: null, key: '7ko70qse'} ⓘ
  hash: ""
  key: "7ko70qse"
  pathname: "/category/Electronics"
  search: "?maxPrice=900"
  state: null
  ► [[Prototype]]: Object
```

```
▼ {pathname: '/category/Electronics', search: '', hash: '#section1', state: null, key: 'default'} ⓘ
  hash: "#section1"
  key: "default"
  pathname: "/category/Electronics"
  search: ""
  state: null
  ► [[Prototype]]: Object
```

**Описание всех свойств объекта useLocation:**

# Router useLocation()

— подробнее про свойство state

## Когда использовать свойство state из useLocation?

Использование state в маршрутах React Router нужно для передачи временных данных между страницами, которые:

1. Не должны отображаться в URL.
2. Не нуждаются в глобальном состоянии (например, через Redux или Context).
3. Должны использоваться только для текущей сессии (данные не сохраняются после перезагрузки страницы).

```
<Link to="/category/Electronics" state={{ from: "Home", maxPrice: 600 }}>
  About
</Link>

const location = useLocation();
const maxPrice = location.state.maxPrice;
```

Взяли значение максимальной цены из свойства state хука useLocation для фильтрации товаров по максимальной цене



# Router **useNavigate()**

— Хук. Программно перенаправляет на указанную страницу  
(ответ на событие или действие)

# useNavigate()

— это хук из библиотеки `react-router-dom`, который позволяет программно осуществлять навигацию между страницами в вашем приложении на React.

Это полезно, когда вам нужно переместить пользователя на другую страницу в ответ на какое-либо действие (например, после отправки формы или нажатия кнопки), без использования компонента `<Link>`.

## Как это работает:

Хук возвращает функцию `navigate`, которую можно вызвать для перехода на другую страницу.

```
import React from 'react';
import { useNavigate } from 'react-router-dom';

function HomePage() {
  const navigate = useNavigate(); // Получаем функцию navigate

  const goToPricing = () => {
    // Перенаправляем пользователя на страницу Pricing
    navigate('/pricing');
  };

  return (
    <div>
      <h1>Welcome to the Home Page!</h1>
      <button onClick={goToPricing}>Go to Pricing</button>
    </div>
  );
}
```

Навигация назад: Можно использовать навигацию назад (например, как кнопка “Назад” в браузере):

```
navigate(-1); // Возвращает на предыдущую страницу
```

Навигация вперед (в истории): Можно перемещаться и вперед по истории, если пользователи уже посетили другие страницы:

```
navigate(1); // Вперёд на одну страницу в истории
```

Замена текущего пути: По умолчанию, `navigate()` добавляет новый элемент в историю браузера, позволяя пользователю вернуться назад. Если нужно заменить текущий путь (чтобы кнопка “Назад” не вернула на предыдущий маршрут), можно передать дополнительный параметр `{ replace: true }`:

```
navigate('/dashboard', { replace: true });
```

Передача состояния: При навигации можно передавать дополнительное состояние через объект:

```
navigate('/profile', { state: { userId: 123 } });
```

Это состояние можно затем получить в целевом компоненте через хук `useLocation`:

```
import { useLocation } from 'react-router-dom';

function Profile() {
  const location = useLocation();
  const { userId } = location.state || {};
  return <div>User ID: {userId}</div>;
}
```

# Router <Navigate />

– Компонент. Программно перенаправляет на указанную страницу (автоматический редирект.)

# Router <Navigate />

– Это компонент React Router, который:

- Перенаправляет пользователя на указанный путь.
- Может быть использован для редиректов (например, после авторизации или на 404-странице).

## Как использовать <Navigate />?

### 1. Простой редирект:

```
import { Navigate } from 'react-router-dom';

function HomePage() {
  const isLoggedIn = false;

  if (!isLoggedIn) {
    return <Navigate to="/login" replace={true} />;
  }

  return <div>Welcome to the Home Page!</div>;
}

export default HomePage;
```

**to="/login":** Указывает путь, на который нужно перенаправить пользователя.

**replace={true}:** Указывает, что текущий маршрут будет заменен в истории браузера (не добавится новый).

### 2. Редирект через маршруты:

Вы можете использовать <Navigate /> в конфигурации маршрутов.

```
import { createBrowserRouter, RouterProvider } from 'react-router-dom';
import Home from './pages/Home';
import Login from './pages/Login';

const router = createBrowserRouter([
  {
    path: '/',
    element: <Home />,
  },
  {
    path: '/login',
    element: <Login />,
  },
  {
    path: '*',
    element: <Navigate to="/" replace={true} />, // Перенаправление на главную страницу
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

Здесь path: "\*" означает, что при любом неизвестном пути пользователь будет перенаправлен на /.

# Router <Navigate />

— Продолжение

## Как использовать <Navigate />?

### 3. Редирект после выполнения действия:

Иногда вам нужно перенаправить пользователя после выполнения какого-либо действия, например, после отправки формы. После успешного входа пользователь перенаправляется на главную страницу (/).

```
import { useState } from 'react';
import { Navigate } from 'react-router-dom';

function LoginPage() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  const handleLogin = () => {
    // Логика авторизации
    setIsLoggedIn(true);
  };

  if (isLoggedIn) {
    return <Navigate to="/" />;
  }

  return (
    <div>
      <h1>Login</h1>
      <button onClick={handleLogin}>Log in</button>
    </div>
  );
}

export default LoginPage;
```

### 4. Передача состояния через <Navigate />:

Вы можете передать данные при навигации с помощью свойства state.

```
import { Navigate } from 'react-router-dom';

function HomePage() {
  const isLoggedIn = false;

  if (!isLoggedIn) {
    return <Navigate to="/login" state={{ message: 'Please log in first' }} />;
  }

  return <div>Welcome to the Home Page!</div>;
}

export default HomePage;
```

Получение состояния на целевой странице:

```
import { useLocation } from 'react-router-dom';

function LoginPage() {
  const location = useLocation();
  const message = location.state?.message || 'Welcome to the login page';

  return <div>{message}</div>;
}

export default LoginPage;
```

# **<Link /> vs <Navigate /> vs useNavigate()**

— Что и когда использовать

# <Link />

<Link> — это компонент React Router, который создаёт навигационную ссылку, позволяющую пользователю перейти на другой маршрут при клике.

Используется для декларативной навигации: вы определяете ссылку в JSX, и она всегда будет вести на указанный маршрут.

## Когда использовать <Link>?

- Когда пользователь должен видеть и кликать на ссылку для перехода.
- Пример: Меню навигации, кнопки перехода, ссылки на другие страницы.

```
import { Link } from "react-router-dom";
function Navigation() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/contact">Contact</Link>
    </nav>
  );
}
```

### Плюсы:

- Простота: добавляет ссылку в DOM.
- Удобен для видимых навигационных элементов.

### Минусы:

- Не подходит для программной навигации (например, при выполнении действия).

# <Navigate />

<Navigate> — это компонент React Router, который позволяет выполнить автоматический редирект.

Когда <Navigate> рендерится, он сразу перенаправляет пользователя на указанный маршрут.

## Когда использовать <Navigate>?

Когда редирект должен происходить условно в зависимости от логики вашего приложения.

Пример: Пользователь не авторизован и должен быть перенаправлен на страницу входа.

```
import { Navigate } from "react-router-dom";

function PrivateRoute({ isAuthenticated }) {
  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }
  return <div>Private Content</div>;
}
```

### Плюсы:

- Удобен для условного редиректа.
- Можно передать состояние (state) при перенаправлении.

### Минусы:

- Используется только в JSX, поэтому не подходит для редиректа из функций или событий.

# useNavigate()

- useNavigate — это хук React Router, который позволяет выполнять программную навигацию (переход на другой маршрут) в ответ на событие или действие пользователя.
- Это замена старого useHistory из React Router v5.

## Когда использовать useNavigate?

• Когда переход должен происходить в ответ на действие пользователя или после выполнения кода.

• Пример: После успешного входа, отправки формы или клика на кнопку.

```
import { useNavigate } from "react-router-dom";

function LoginPage() {
  const navigate = useNavigate();

  const handleLogin = () => {
    // Логика авторизации
    navigate("/dashboard"); // Программный редирект
  };

  return <button onClick={handleLogin}>Log In</button>;
}
```

### Плюсы:

- Идеален для программной навигации.
- Можно использовать в любой функции или событии.

### Минусы:

- Не подходит для декларативной навигации, так как не является частью JSX.

# CSS Modules

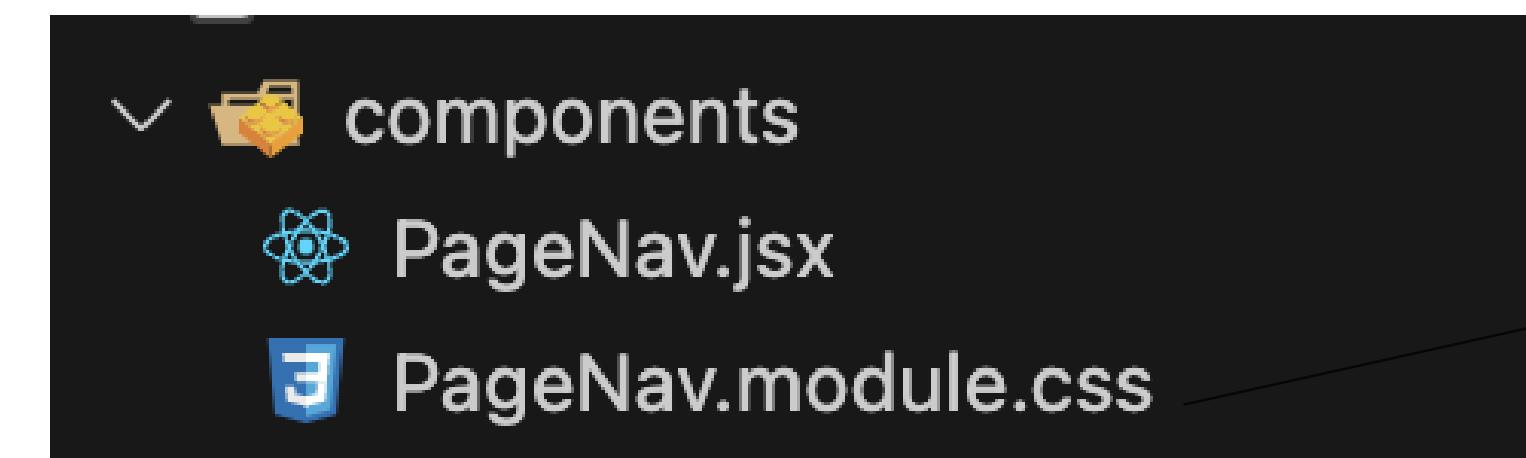
— как работают

# CSS Modules

— это техника и набор инструментов для работы с CSS в JavaScript-проектах (например, в React), которая позволяет автоматически изолировать стили, чтобы избежать конфликтов имен классов и влияния стилей на другие компоненты.

## Пример использования CSS Modules

Создаем отдельный файл со стилями для каждого компонента и прописываем в нем стили



```
.nav {
  background-color: red;
}

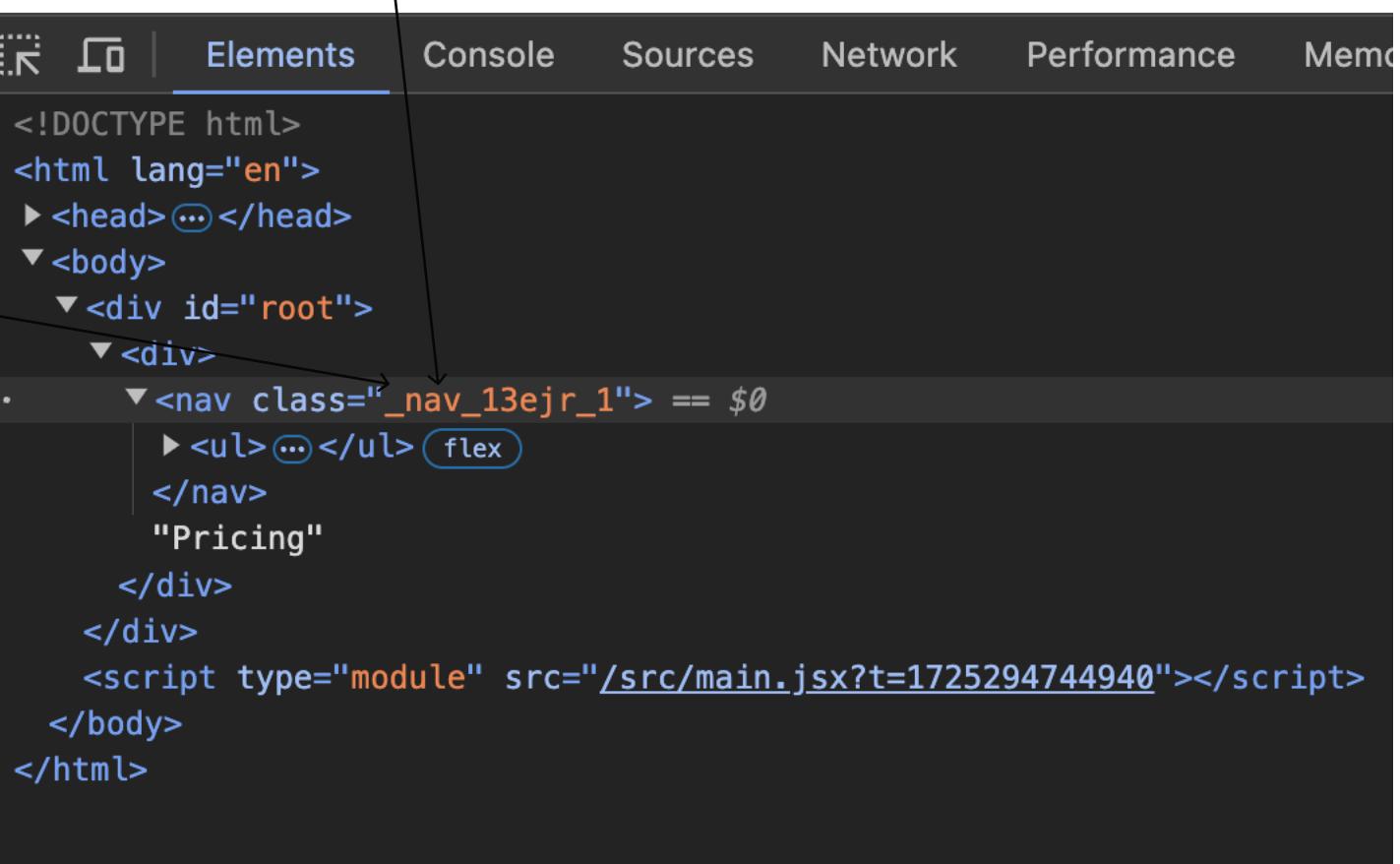
.nav ul {
  display: flex;
  justify-content: space-between;
}
```

Импортируем стили в компонент и используем в элементе:

```
import { NavLink } from "react-router-dom";
import styles from "./PageNav.module.css";

function PageNav() {
  return (
    <nav className={styles.nav}>
      <ul>
        <li>
          <NavLink to="/">Home Page</NavLink>
        </li>
        <li>
          <NavLink to="/pricing">Pricing</NavLink>
        </li>
        <li>
          <NavLink to="/product">Product</NavLink>
        </li>
      </ul>
    </nav>
  );
}

export default PageNav;
```



Класс применяется модифицированным. Так поддерживается уникальность классов

## CSS Modules global

– CSS Modules поддерживают возможность создания локальных стилей с вложенностью, а также определение глобальных стилей, которые могут быть использованы во всем приложении.

**Пример:**

```
/* styles.module.css */
:global(.global-class) {
    background-color: red;
}
.local-class {
    background-color: green;
}
```

React Rout до 6.4

# Data Loading

– Загрузка данных в React Rout (Старый подход)

# Rout Data Loading до версии 6.4

До версии 6.4 в React Router  
работа с данными в маршрутах  
(routes) не включала  
встроенные механизмы для  
загрузки данных напрямую в  
маршруты. Вместо этого  
разработчики использовали  
хуки, такие как useEffect, и  
управляли состоянием данных  
вручную внутри компонентов.

— Можете скопировать код из  
комментариев ниже и протестировать

## Процесс работы с данными

Обычная функция fetch к API

```
// Function to fetch data
const fetchData = async () => {
  const response = await fetch("https://jsonplaceholder.typicode.com/posts");
  if (!response.ok) {
    throw new Error("Failed to fetch posts");
  }
  return response.json();
};
```

## Страница постов

```
function Posts() {
  const [posts, setPosts] = useState([]);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const loadPosts = async () => {
      try {
        const data = await fetchData();
        setPosts(data);
      } catch (err) {
        setError(err);
      } finally {
        setLoading(false);
      }
    };
    loadPosts();
  }, []);

  if (loading) {
    return <p>Loading...</p>;
  }

  if (error) {
    return (
      <div>
        <h1>Error</h1>
        <p>{error.message || "Something went wrong"}</p>
      </div>
    );
  }

  return (
    <div>
      <h1>Posts</h1>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>{post.title}</li>
        )));
      </ul>
      <Link to="/">Go back to Home</Link>
    </div>
  );
}
```

## Структура сайта

```
// Define routes
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />, // Home page
  },
  {
    path: "/posts",
    element: <Posts />, // Posts page
  },
  {
    path: "*",
    element: <h1>404: Page Not Found</h1>, // Component for non-existent routes
  },
]);
```

### 1. Первичный рендеринг и отображение страницы:

- Когда пользователь переходит на страницу, React выполняет рендеринг компонента. При этом отрисовываются базовые элементы компонента (например, заголовки, статический текст) до загрузки данных.

### 2. Запуск useEffect:

- После первого рендеринга запускается useEffect, так как он по умолчанию выполняется после монтирования компонента.

### 3. Загрузка данных:

- Внутри useEffect происходит асинхронная загрузка данных (например, с использованием fetch), и по завершении данные сохраняются в состоянии (state) с помощью функции, возвращаемой useState.

### 4. Повторный рендеринг:

- Когда состояние обновляется (в данном случае, данные, полученные из API), React вызывает повторный рендеринг компонента для обновления отображения с новыми данными.

# React Router v6.4+

## **loading**

- Параметр маршрута для загрузки данных из сервера  
(Современный подход)

# Rout 6.4 + Data Loading

— это новая версия библиотеки для маршрутизации в React-приложениях.

Она значительно упрощает управление маршрутами и включает множество новых возможностей по сравнению с предыдущими версиями, таких как декларативное управление маршрутизацией, поддержка асинхронных данных через loader и action, а также улучшенная работа с вложенными маршрутами.

## Основные изменения в React Router v6.4:

1. Поддержка загрузки данных с помощью loader.
2. Обработка данных форм через action.
3. Централизованная обработка ошибок через errorElement.
4. Улучшенная работа с асинхронными операциями.
5. Новые хуки для отслеживания состояния навигации (useNavigation) и обработки ошибок (useRouteError).

# Rout 6.4

## Data Loading (loading)

— Новый способ подгрузки  
данных

— Можете скопировать код из  
комментариев ниже и протестировать

### loader – Загрузка данных перед рендерингом

loader – это функция, которая вызывается перед рендерингом компонента и используется для загрузки данных (например, запрос к API или работа с локальными данными).

Результаты этой функции передаются компоненту через хук useLoaderData.

```
import { createBrowserRouter, RouterProvider, useLoaderData } from "react-router-dom";

// Функция для загрузки данных
async function fetchProduct({ params }) {
  const response = await fetch(`/api/products/${params.productId}`);
  return response.json();
}

// Компонент для отображения данных
function Product() {
  const product = useLoaderData(); // Получаем данные, загруженные через loader
  return <h1>{product.name}</h1>;
}

// Настройка маршрута
const router = createBrowserRouter([
  {
    path: "/product/:productId",
    element: <Product />,
    loader: fetchProduct, // Указываем функцию loader
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

- loader выполняет асинхронную операцию (например, загрузку данных с сервера) перед рендерингом компонента.
- Данные, полученные в loader, передаются компоненту через хук useLoaderData.
- Если во время загрузки данных возникает ошибка, она может быть обработана через errorElement (о нём далее).

React Router v6.4+

# **errorElement**

– Параметр маршрута для обработки ошибок на странице

# Rout 6.4 Data Loading ( errorElement )

— это компонент, который рендерится, если при загрузке данных, отправке формы или рендеринге компонента возникает ошибка.

## errorElement — Обработка ошибок маршрута

errorElement — это компонент, который рендерится, если при загрузке данных, отправке формы или рендеринге компонента возникает ошибка. Он помогает создавать пользовательские страницы с ошибками.

```
import { createBrowserRouter, RouterProvider, useRouteError } from "react-router-dom";

// Компонент для отображения ошибки
function ErrorPage() {
  const error = useRouteError(); // Получаем информацию об ошибке
  return (
    <div>
      <h1>Oops! Something went wrong.</h1>
      <p>{error.statusText || error.message}</p>
    </div>
  );
}

// Функция loader, которая может выбросить ошибку
async function fetchProduct({ params }) {
  const response = await fetch(`/api/products/${params.productId}`);
  if (!response.ok) {
    throw new Error("Failed to fetch product");
  }
  return response.json();
}

// Настройка маршрута с обработкой ошибок
const router = createBrowserRouter([
  {
    path: "/product/:productId",
    element: <Product />,
    loader: fetchProduct,
    errorElement: <ErrorPage />, // Указываем компонент для обработки ошибок
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

- Если в loader или action возникает ошибка, компонент errorElement автоматически рендерится вместо основного компонента.
- Ошибки можно получить с помощью хука useRouteError().

# errorElement 6.4 +

— Работа по шагам

## Как работает errorElement?

1. Когда происходит ошибка в маршруте, React Router автоматически заменяет стандартный контент на компонент, указанный в errorElement.
2. Ошибка передаётся в компонент через проп error.

### Функция fetch для запросом к данным

```
// Function to fetch data
const fetchData = async () => {
  const response = await fetch("https://jsonplaceholder.typicode.com/posts");
  if (!response.ok) {
    throw new Error("Failed to fetch posts");
  }
  return response.json();
};
```

### Страница которая использует данные

```
// Component for the Posts page
function Posts() {
  const posts = useLoaderData(); // Use data loaded by the loader
  return (
    <div>
      <h1>Posts</h1>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
      <Link to="/">Go back to Home</Link>
    </div>
  );
}
```

### Страница которая отображается при ошибке

```
// Component for handling errors
function ErrorBoundary({ error }) {
  return (
    <div>
      <h1>Error</h1>
      <p>{error.message || "Something went wrong"}</p>
    </div>
  );
}
```

— Можете скопировать код из  
комментариев ниже и протестировать

### Структура приложения

```
// Define routes
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />, // Home page
  },
  {
    path: "/posts",
    element: <Posts />, // Posts page
    loader: fetchData, // Data loader
    errorElement: <ErrorBoundary />, // Error handler
  },
  {
    path: "*",
    element: <h1>404: Page Not Found</h1>, // Component for non-existent routes
  },
]);
```

# useRouteError()

— Хук useRouteError()  
применяется в errorElement  
компоненте маршрута для  
получения информации об  
ошибке, возникшей при  
загрузке данных (loader) или  
действиях (action), а также при  
выполнении какого-либо кода  
в процессе рендеринга  
текущего маршрута.

## Как работает useRouteError()

1-Создаем запрос к серверу и выбрасываем ошибку при неверном ответе сервера.

```
// Function to fetch data
const fetchData = async () => {
  const response = await fetch("https://jsonplaceholder.typicode.com/postss");
  if (!response.ok) {
    throw new Error("Failed to fetch posts");
  }
  return response.json();
};
```

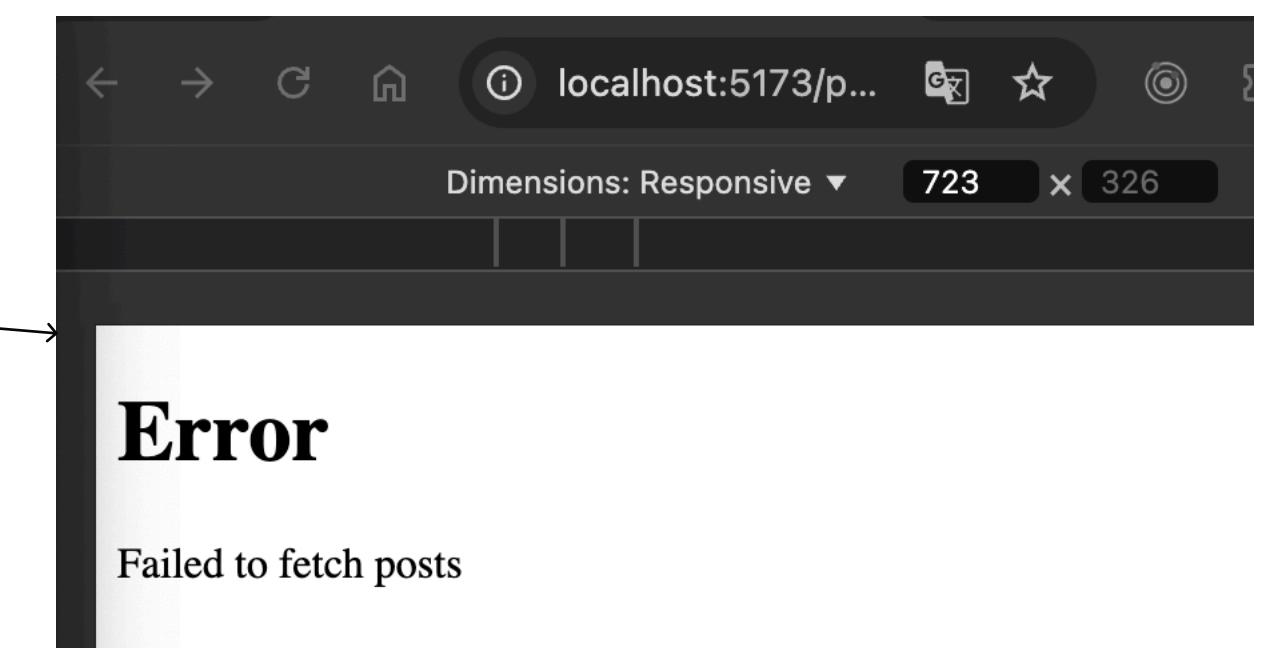
2-Определяем компонент, который будет отрендерен в  
случае ошибки.

```
// Define routes
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />, // Home page
  },
  {
    path: "/posts",
    element: <Posts />, // Posts page
    loader: fetchData, // Data loader
    errorElement: <ErrorBoundary />, // Error handler
  },
  {
    path: "*",
    element: <h1>404: Page Not Found</h1>, // Component for non-existent routes
  },
]);
```

Хук useRouteError() позволяет получить  
объект ошибки, возникающий при вып  
олнении запроса fetch.

3-Объявляем компонент для отображения на экране при  
возникновении ошибки.

```
// Component for handling errors
function ErrorBoundary() {
  const error = useRouteError();
  return (
    <div>
      <h1>Error</h1><p>{error.message || "Something went wrong"}</p>
    </div>
  );
}
```



# React Router v6.4+ **(useNavigation)**

— хук `useNavigation`, который позволяет отслеживать состояние текущей навигации

# Rout 6.4 Хук (useNavigation)

— хук useNavigation, который позволяет отслеживать состояние текущей навигации

## Отслеживание состояния навигации с useNavigation

useNavigation возвращает объект с состоянием текущей навигации (idle, submitting, loading). Это полезно для отображения сообщений о загрузке или блокировки интерфейса при выполнении асинхронных операций.

- **idle** – нет активной навигации или загрузки.
- **loading** – происходит загрузка данных для нового маршрута.
- **submitting** – выполняется отправка формы или другое действие.

## Использование useLoaderData и useParams

Создайте компонент, отображающий

```
function Layout() {
  const navigation = useNavigation();

  return (
    <div>
      {navigation.state === "loading" && <p>Loading...</p>}
      <Outlet />
    </div>
  );
}
```

Оберните в этот компонент все приложение

```
// Defining routes
const router = createBrowserRouter([
  {
    element: <Layout />,
    children: [
      {
        path: "/",
        element: <Home />, // Home page
      },
      {
        path: "/posts",
        element: <Posts />, // Posts page
        loader: fetchData, // Specify the loader function
        errorElement: <ErrorBoundary />, // Error handler
      },
    ],
  },
  {
    path: "*",
    element: <h1>404: Page Not Found</h1>, // Component for non-existent routes
  },
]);

// Main application component
function App() {
  return <RouterProvider router={router}>;
}

export default App;
```

# React Router v6.4+

## **action**

- Параметр маршрута для отправки данных на сервер

# Rout 6.4

## Параметр (Action)

— ШАГИ ПО ИСПОЛЬЗОВАНИЯ

— Можете скопировать код из  
комментариев ниже и протестировать

## action – Обработка данных форм

action используется для обработки изменений данных (например, отправка данных формы методом POST, PUT, DELETE и т.д.). Когда форма отправляется, функция action вызывается для выполнения операции на сервере (например, создание или обновление записи).

```
import { createBrowserRouter, RouterProvider, Form } from "react-router-dom";
// Функция для обработки отправки формы
async function createOrder({ request }) {
  const formData = await request.formData();
  const orderDetails = Object.fromEntries(formData); // Преобразуем данные формы в объект

  // Отправляем данные на сервер
  await fetch("/api/orders", {
    method: "POST",
    body: JSON.stringify(orderDetails),
    headers: { "Content-Type": "application/json" },
  });
}

// Компонент с формой
function CreateOrder() {
  return (
    <Form method="POST"> /* Форма автоматически отправит данные через action */
      <input type="text" name="customerName" required placeholder="Customer Name" />
      <button type="submit">Create Order</button>
    </Form>
  );
}

// Настройка маршрута
const router = createBrowserRouter([
  {
    path: "/create-order",
    element: <CreateOrder />,
    action: createOrder, // Указываем функцию action
  },
]);
function App() {
  return <RouterProvider router={router} />;
}
export default App;
```

### 1. Форма (Form) в компоненте отправляет данные:

- Когда пользователь заполняет и отправляет форму, она отправляет данные на сервер (или обрабатывается локально).
- В React Router формы можно сделать декларативными с помощью компонента <Form>.

### 2. action обрабатывает запрос:

- Когда форма отправляется, React Router вызывает функцию action, связанную с маршрутом, к которому привязана форма.
- Функция action получает объект, содержащий request и params.

### 3. Обработка данных:

- Внутри action вы можете:
- Получить данные из запроса (request).
- Извлечь параметры маршрута (params).
- Выполнить запрос к API, обновить базу данных или локальное состояние.
- Вернуть результат, который будет доступен через useActionData.

### 4. Результат action:

- Данные, возвращаемые из action, доступны в компоненте с помощью хука useActionData.

- action вызывается при отправке формы. Форма должна использовать метод POST, PUT, или DELETE.
- Функция action получает объект request, который содержит данные формы, и отправляет их на сервер.
- Можно выполнять любую асинхронную операцию в action (например, запрос к API, сохранение данных).

# Context API

– Решение проблемы проброса пропсов

# Context API

— предназначен для решения проблемы “проброса пропсов” (props drilling) — когда данные передаются через несколько уровней дерева компонентов от родителя к дочерним компонентам.

Context API позволяет делиться данными напрямую между компонентами на любом уровне дерева, избегая необходимости передавать их через каждый уровень.

## Основные компоненты Context API:

1. Создание контекста (createContext)
2. Провайдер (Provider)
3. Потребители контекста (useContext и Consumer)

### 1. Создание контекста

Контекст создается с помощью функции createContext(). Это создает объект контекста, который содержит два компонента:

- Provider — предоставляет данные.
- Consumer — получает данные (чаще всего используется useContext, вместо Consumer).

Функция создается в снаружи компонента

```
const MyContext = React.createContext();
```

Функция возвращает компонент. Поэтому название функции с большой буквы

### 2. Провайдер контекста (Provider)

Провайдер — это компонент, который обворачивает дерево компонентов и предоставляет данные, которые могут быть использованы любым компонентом, находящимся внутри. Provider принимает специальный проп value, который содержит данные для передачи. Для использования — обворачиваем JSX в return родительского компонента в компонент MyContext.Provider

```
<MyContext.Provider value={/* данные */}>
  {/* дочерние компоненты */}
</MyContext.Provider>
```

### 3. Потребители контекста

хук useContext позволяет достать данные контекста в любом компоненте, который находится внутри Provider.

```
import { useContext } from 'react';
const value = useContext(MyContext);
```

# useReducer()

— XYK

# useReducer()

— это хук в React, который позволяет управлять сложным состоянием компонента. Он похож на useState, но предоставляет больше возможностей для организации и управления состоянием, особенно если изменения состояния зависят от различных типов действий.

```
import React, { useReducer } from 'react';

// Определение функции-редюсера
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    case 'reset':
      return 0;
    default:
      throw new Error('Unknown action type');
  }
}

function Counter() {
  // Использование useReducer с начальным значением 0
  const [count, dispatch] = useReducer(reducer, 0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </div>
  );
}
```

**useReducer принимает два аргумента:**

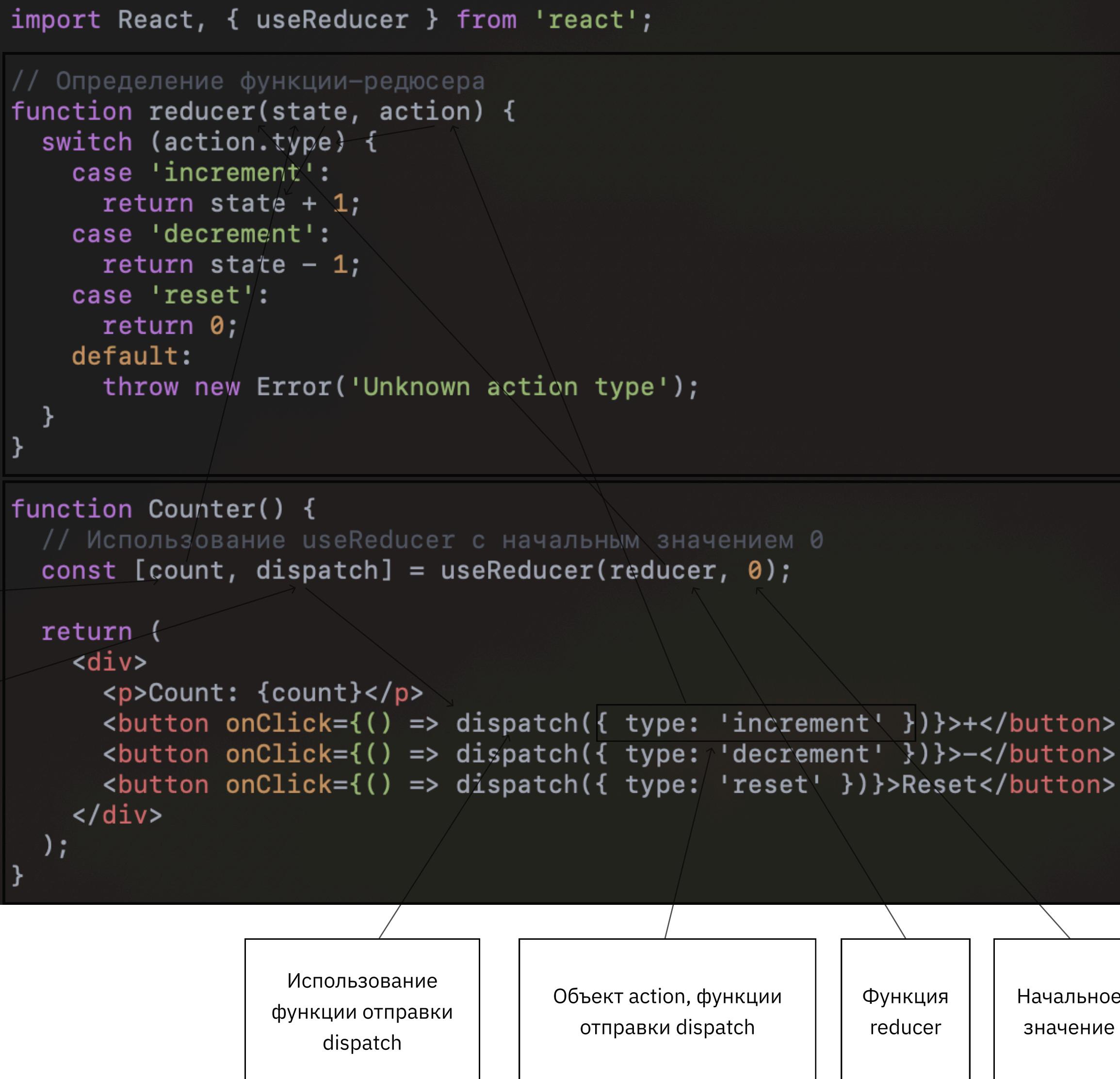
1. reducer: функция-редюсер, определенная выше(вне компонента).
2. { count: 0 }: начальное состояние.

**Возвращает массив из двух элементов:**

1. state: текущее состояние (в данном случае — число 0).
2. dispatch: функция для отправки действий в функцию редюсер в качестве аргумента.

## Схема работы useReducer()

Функция  
снаружи  
компонента



— Можете скопировать код из  
комментариев ниже и протестировать

## Работа по шагам:

### 1. Рендеринг начального состояния:

- При первом рендре useReducer инициализирует состояние { count: 0 }, и компонент отображает Count: 0.

### 2. Обработка нажатий на кнопки:

- Когда пользователь нажимает кнопку +, выполняется функция dispatch({ obj}).

Функция dispatch отправляет объект “действие(action)” во внешнюю функцию редюсер, которая берет значение свойства type объекта action и проверяет его в switch.

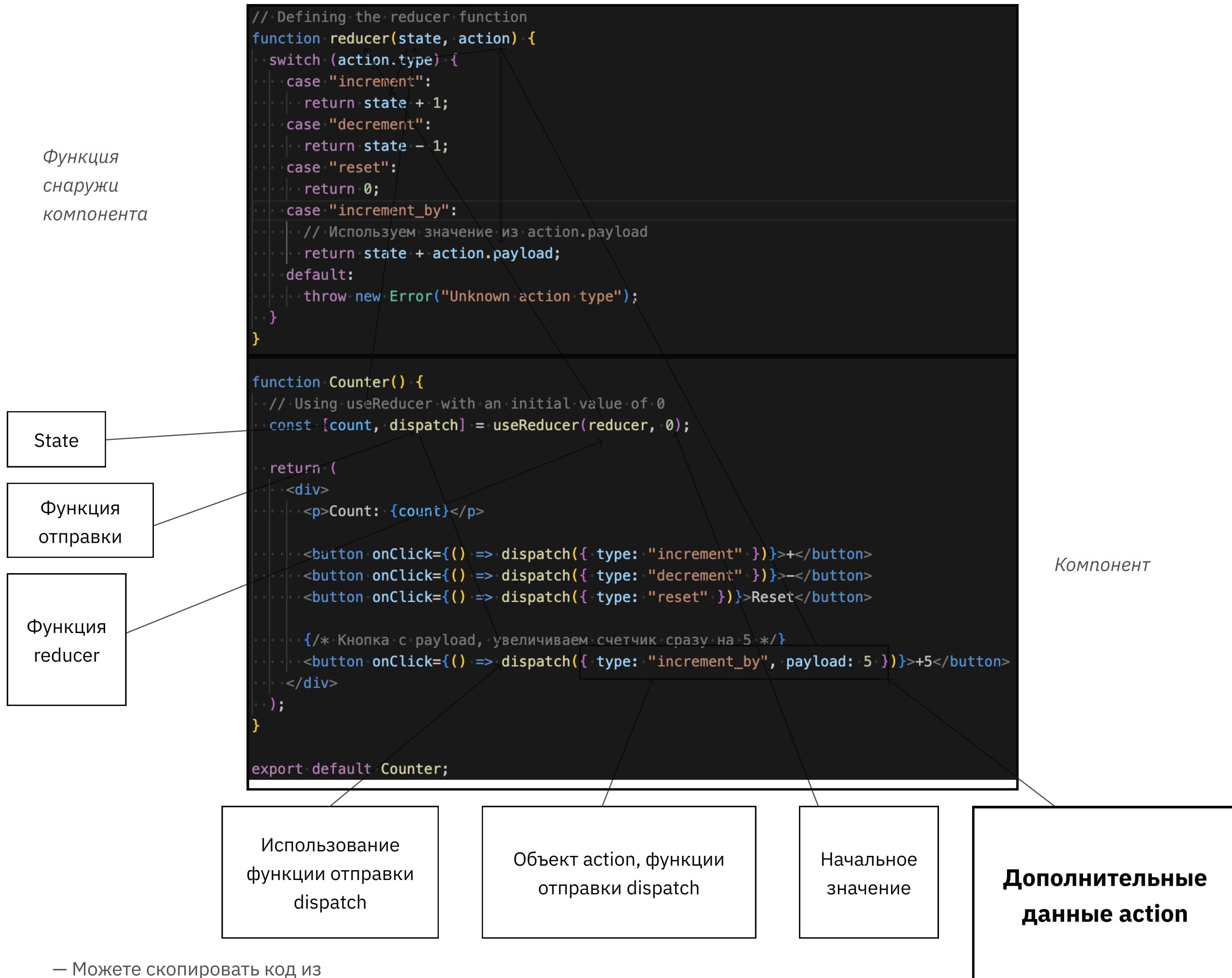
В случае если action.type == “increment” увеличивает состояние на 1, в случае “decrement” уменьшает на 1, в случае “reset” сбрасывает count на 0

### 3. Обновление компонента:

- После каждого вызова dispatch React перерисовывает компонент с новым состоянием, и на экране обновляется значение count.

useReducer() payload  
– action.payload

# Схема работы useReducer()



## Работа по шагам:

### 1. Рендеринг начального состояния:

- При первом рендрере useReducer инициализирует состояние { count: 0 }, и компонент отображает Count: 0.

### 2. Обработка нажатий на кнопки:

- Когда пользователь нажимает кнопку +, выполняется функция dispatch({ obj}).

Функция dispatch отправляет объект “действие(action)” во внешнюю функцию редюсер, которая берет значение свойства type объекта action и проверяет его в switch.

В случае если action.type == “increment” увеличивает состояние на 1,  
в случае “decrement” уменьшает на 1,  
в случае “reset” сбрасывает count на 0

В случае нажатия на кнопку +5 state увеличивается на число прописанное в action.payload

### 3. Обновление компонента:

- После каждого вызова dispatch React перерисовывает компонент с новым состоянием, и на экране обновляется значение count.

payload — это «дополнительная информация» о том, что конкретно нужно сделать с состоянием.

— Можете скопировать код из  
комментариев ниже и протестировать

useReducer() initialState  
+ e.target.value

## Начальное состояние в виде объекта, а не примитива

```
// 1. Combine all required data into a single object
const initialState = {
  count: 0, // Stores the current value of the counter
  inputValue: "", // Stores the current value entered in the input field
};
```

## Функция reducer для управления состояниями

```
// 2. Update the reducer
function reducer(state, action) {
  switch (action.type) {
    case "increment":
      // Increase the counter by 1
      return { ...state, count: state.count + 1 };
    case "decrement":
      // Decrease the counter by 1
      return { ...state, count: state.count - 1 };
    case "reset":
      // Reset the counter to 0
      return { ...state, count: 0 };
    case "update_input":
      // Update the inputValue with the value from action.payload
      return { ...state, inputValue: action.payload };
    case "increment_by":
      // Add the value from payload to count
      return { ...state, count: state.count + action.payload };
    default:
      throw new Error("Unknown action type");
  }
}
```

## Компонент

```
function Counter() {
  // Now both count and inputValue are stored in state
  const [state, dispatch] = useReducer(reducer, initialState);

  const handleIncrementBy = () => {
    const value = parseInt(state.inputValue, 10);
    if (!isNaN(value)) {
      dispatch({ type: "increment_by", payload: value });
    }
    // Clear the input field
    dispatch({ type: "update_input", payload: "" });
  };

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: "increment" })}>+</button>
      <button onClick={() => dispatch({ type: "decrement" })}>-</button>
      <button onClick={() => dispatch({ type: "reset" })}>Reset</button>

      <div>
        <input
          type="number"
          value={state.inputValue}
          onChange={(e) => dispatch({ type: "update_input", payload: e.target.value })}
          placeholder="Enter a number"
        />
        <button onClick={handleIncrementBy}>Increase by</button>
      </div>
    </div>
  );
}

export default Counter;
```

## Input:

### 1. Событие onChange:

Когда пользователь вводит значение в поле, событие onChange срабатывает и вызывает функцию dispatch.

- Тип действия: "update\_input".

- Значение, введённое в поле, передаётся через action.payload.

### 2. Привязка состояния value:

Поле ввода (input) связывается с состоянием state.inputValue через value.

Это означает, что каждый раз, когда state.inputValue обновляется, значение поля автоматически синхронизируется.

### 3. Очистка поля:

После нажатия на кнопку Increase by, inputValue очищается, чтобы пользователь видел пустое поле. Это делается через dispatch с типом "update\_input" и пустым значением (payload: "").

## initialState :

Объект initialState определяет начальные значения всех переменных состояния. Он используется как базовое состояние при первом рендеринге компонента:

## Более реальный пример использования useReducer()

```
1 import { useReducer, useState } from "react";
2
3 function reducer(state, action) {
4   if (action.type === "inc") return { ...state, count: state.count + state.step };
5   if (action.type === "dec") return { ...state, count: state.count - state.step };
6   if (action.type === "res") return { count: 0, step: 1 };
7   if (action.type === "setCount") return { ...state, count: action.payload };
8   if (action.type === "setStep") return { ...state, step: action.payload };
9   return state;
10 }
11
12 function DateCounter() {
13   const initialState = { count: 0, step: 1 };
14   const [state, dispatch] = useReducer(reducer, initialState);
15
16 // This mutates the date object.
17 const date = new Date("june 21 2027");
18 date.setDate(date.getDate() + state.count);
19
20 const inc = function () {
21   dispatch({ type: "inc" });
22 };
23
24 const dec = function () {
25   dispatch({ type: "dec" });
26 };
27
28 const res = function () {
29   dispatch({ type: "res" });
30 };
31
32 const defineCount = function (e) {
33   dispatch({ type: "setCount", payload: Number(e.target.value) });
34 };
35
36 const defineStep = function (e) {
37   dispatch({ type: "setStep", payload: Number(e.target.value) });
38 };
39
40 return (
41   <div>
42     <div>
43       <input type="range" min="0" max="10" value={state.step} onChange={defineStep}>
44       <span>{state.step}</span>
45     </div>
46
47     <div>
48       <button onClick={dec}>-</button>
49       <input value={state.count} onChange={defineCount}>
50       <button onClick={inc}>+</button>
51     </div>
52
53     <p>{date.toDateString()}</p>
54
55     <div>
56       <button onClick={res}>Reset</button>
57     </div>
58   </div>
59 );
60
61 export default DateCounter;
```

Деструктуризация  
объекта и изменение  
одного из свойств

Переменная с  
**начальным**  
состоянием

Начальное  
состояние **объект**,  
а не примитив

**payload** общепринятое  
название для свойства которое  
работает при изменении  
объекта в функции reducer

Используем отдельную  
функцию, в которой работает  
dispatch.

# Redux

– Управление состояниями в большом проекте

# Redux

— по своей структуре очень похож на работу хука useReducer()

## Основные концепции (термины/ блоки) Redux

1. **Store (хранилище)** – единый объект состояния для всего приложения. Он содержит всё состояние приложения.
2. **Actions (действия)** – объекты, которые описывают события или изменения, которые должны произойти в состоянии.
3. **Reducers (редьюсеры)** – чистые функции, которые принимают текущее состояние и действие, а затем возвращают новое состояние.
4. **Dispatch (отправка действий)** – способ отправить действие в хранилище, чтобы редьюсер обработал его и обновил состояние.
5. **Selectors (селекторы)** – функции, которые извлекают нужные данные из состояния для использования в компонентах.

## Основные шаги по созданию и использованию Redux

### 1. Установка Redux и React-Redux:

Для начала нужно установить библиотеки redux и react-redux:

```
npm install redux react-redux
```

### 2. Создание Reducer (редьюсера)

Reducer – это чистая функция, которая принимает текущее состояние и действие, и возвращает новое состояние на основе типа действия. (Так же как в useReducer)

```
// reducers.js
const initialState = {
  balance: 0,
};

export const accountReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'account/deposit':
      return {
        ...state,
        balance: state.balance + action.payload,
      };
    case 'account/withdraw':
      return {
        ...state,
        balance: state.balance - action.payload,
      };
    default:
      return state;
  }
};
```

# Redux

— шаги по созданию и  
использования

## 3. Создание Store (хранилища)

Store — это центральное хранилище состояния приложения. Оно создается с помощью функции createStore.

```
// store.js
import { createStore } from 'redux';
import rootReducer from './rootReducer';

// Создаем хранилище с редьюсерами
const store = createStore(rootReducer);

export default store;
```

## 4. Подключение Redux к React

Для подключения React к Redux используется компонент Provider из библиотеки react-redux. Он предоставляет доступ к хранилищу всем компонентам приложения.

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './App';
import store from './store';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

## 5. Создание Action (действий)

Actions — это объекты, которые передаются в хранилище, чтобы вызвать изменение состояния.

У них всегда есть поле type и иногда дополнительная информация в payload. (Так же как в useReducer)

```
// actions.js
// Действие для пополнения баланса
export const deposit = (amount) => {
  return {
    type: 'account/deposit',
    payload: amount,
  };
};

// Действие для снятия средств
export const withdraw = (amount) => {
  return {
    type: 'account/withdraw',
    payload: amount,
  };
};
```

# Redux

— шаги по созданию и  
использования

## 6. Доступ к состоянию в компонентах с использованием useSelector

Чтобы получить доступ к данным из хранилища Redux в компоненте, используется хук useSelector.

```
// Balance.js
import React from 'react';
import { useSelector } from 'react-redux';

const Balance = () => {
  const balance = useSelector((state) => state.account.balance);

  return <div>Current Balance: {balance}</div>;
};

export default Balance;
```

## 7. Отправка действий с помощью useDispatch

Чтобы изменить состояние, используется хук useDispatch, который позволяет отправить действие в хранилище.

```
// AccountActions.js
import React from 'react';
import { useDispatch } from 'react-redux';
import { deposit, withdraw } from './actions';

const AccountActions = () => {
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() => dispatch(deposit(100))}>Deposit $100</button>
      <button onClick={() => dispatch(withdraw(50))}>Withdraw $50</button>
    </div>
  );
};

export default AccountActions;
```

# Redux

— шаги по созданию и  
использования

## 8. Комбинирование редьюсеров (combineReducers)

Если приложение имеет несколько независимых частей состояния, можно использовать функцию combineReducers, чтобы объединить несколько редьюсеров.

```
// rootReducer.js

import { combineReducers } from 'redux';
import { accountReducer } from './reducers';

const rootReducer = combineReducers({
  account: accountReducer,
  // можно добавить больше редьюсеров, если они нужны
});

export default rootReducer;
```

## 9. Просмотр текущего состояния

Для просмотра текущего состояния приложения (например, в отладочных целях), можно использовать метод getState из хранилища:

```
console.log(store.getState());
```

## Полный пример:

```
// actions.js
export const deposit = (amount) => ({ type: 'account/deposit', payload: amount });
export const withdraw = (amount) => ({ type: 'account/withdraw', payload: amount });

// reducers.js
const initialState = { balance: 0 };

export const accountReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'account/deposit':
      return { ...state, balance: state.balance + action.payload };
    case 'account/withdraw':
      return { ...state, balance: state.balance - action.payload };
    default:
      return state;
  }
};

// rootReducer.js
import { combineReducers } from 'redux';
import { accountReducer } from './reducers';
export const rootReducer = combineReducers({ account: accountReducer });

// store.js
import { createStore } from 'redux';
import { rootReducer } from './rootReducer';
export const store = createStore(rootReducer);

// App.js
import React from 'react';
import Balance from './Balance';
import AccountActions from './AccountActions';

const App = () => (
  <div>
    <h1>Redux Bank</h1>
    <Balance />
    <AccountActions />
  </div>
);

export default App;

// Balance.js
import React from 'react';
import { useSelector } from 'react-redux';

const Balance = () => {
  const balance = useSelector((state) => state.account.balance);
  return <div>Current Balance: {balance}</div>;
};

export default Balance;

// AccountActions.js
import React from 'react';
import { useDispatch } from 'react-redux';
import { deposit, withdraw } from './actions';

const AccountActions = () => {
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() => dispatch(deposit(100))}>Deposit $100</button>
      <button onClick={() => dispatch(withdraw(50))}>Withdraw $50</button>
    </div>
  );
};

export default AccountActions;
```

# Redux Thunk

– Асинхронность

# Redux Thunk

— это middleware для Redux, который позволяет писать асинхронные действия. Он предоставляет возможность создавать экшены (actions) в виде функций, которые могут выполнять асинхронные запросы и затем диспатчить (отправлять) обычные экшены с результатом этих запросов.

## Основные концепции (термины/ блоки) Redux

**Асинхронные экшены:** redux-thunk позволяет создавать действия в виде функций вместо обычных объектов. Эти функции могут выполнять любые асинхронные операции (например, запросы на сервер).

**Доступ к dispatch и getState:** Внутри функции-экшена, которую предоставляет Thunk, доступны два аргумента: dispatch для отправки действий и getState для получения текущего состояния хранилища.

## Основные шаги по использованию Redux Thunk:

### 1. Установка Redux Thunk

Сначала нужно установить библиотеку:

```
npm install redux-thunk
```

### 2. Подключение Thunk к хранилищу

Thunk нужно подключить как middleware при создании хранилища Redux с помощью функции applyMiddleware.

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './rootReducer';

const store = createStore(rootReducer, applyMiddleware(thunk));

export default store;
```

# Redux Thunk

— шаги по созданию и  
использования

### 3. Создание асинхронных экшенов

Теперь можно создавать асинхронные действия, используя Thunk. Вместо возвращения объекта с полем type, можно возвращать функцию, которая будет выполнять асинхронные запросы и затем диспатчить экшены.

```
// actions.js

// Асинхронное действие для получения данных
export const fetchUserData = () => {
  return async (dispatch) => {
    dispatch({ type: 'user/fetchStart' }); // Отправляем действие начала загрузки

    try {
      const response = await fetch('https://api.example.com/user');
      const data = await response.json();

      dispatch({ type: 'user/fetchSuccess', payload: data }); // Успешный результат
    } catch (error) {
      dispatch({ type: 'user/fetchError', payload: error.message }); // Обработка ошибки
    }
  };
};
```

### 4. Обработка состояний в Reducer

Для обработки состояний загрузки, успеха или ошибки в редьюсере нужно добавить новые типы действий:

```
// reducers.js

const initialState = {
  loading: false,
  user: null,
  error: null,
};

export const userReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'user/fetchStart':
      return {
        ...state,
        loading: true,
        error: null,
      };

    case 'user/fetchSuccess':
      return {
        ...state,
        loading: false,
        user: action.payload,
      };

    case 'user/fetchError':
      return {
        ...state,
        loading: false,
        error: action.payload,
      };

    default:
      return state;
  }
};
```

# Redux Thunk

— шаги по созданию и  
использования

## 5. Диспатч асинхронного экшена из компонента

Теперь можно использовать асинхронное действие в компоненте через хук `useDispatch`.

```
// UserProfile.js

import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { fetchUserData } from './actions';

const UserProfile = () => {
  const dispatch = useDispatch();
  const { user, loading, error } = useSelector((state) => state.user);

  useEffect(() => {
    dispatch(fetchUserData()); // Запрос данных при монтировании компонента
  }, [dispatch]);

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div>
      <h1>{user.name}</h1>
      <p>Email: {user.email}</p>
    </div>
  );
}

export default UserProfile;
```

## 6. Использование `getState`

Внутри асинхронного действия можно использовать `getState`, чтобы получить текущее состояние приложения и использовать его для принятия решений.

```
// actions.js

export const fetchIfNotLoaded = () => {
  return (dispatch, getState) => {
    const state = getState();
    if (!state.user.user) {
      dispatch(fetchUserData()); // Запрос данных, если они ещё не загружены
    }
  };
}
```

## Полный пример:

```
// actions.js
export const fetchUserData = () => {
  return async (dispatch) => {
    dispatch({ type: 'user/fetchStart' });

    try {
      const response = await fetch('https://api.example.com/user');
      const data = await response.json();
      dispatch({ type: 'user/fetchSuccess', payload: data });
    } catch (error) {
      dispatch({ type: 'user/fetchError', payload: error.message });
    }
  };
};

// reducers.js
const initialState = { loading: false, user: null, error: null };

export const userReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'user/fetchStart':
      return { ...state, loading: true, error: null };
    case 'user/fetchSuccess':
      return { ...state, loading: false, user: action.payload };
    case 'user/fetchError':
      return { ...state, loading: false, error: action.payload };
    default:
      return state;
  }
};

// rootReducer.js
import { combineReducers } from 'redux';
import { userReducer } from './reducers';

export const rootReducer = combineReducers({ user: userReducer });

// store.js
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import { rootReducer } from './rootReducer';

const store = createStore(rootReducer, applyMiddleware(thunk));
export default store;

// UserProfile.js
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { fetchUserData } from './actions';

const UserProfile = () => {
  const dispatch = useDispatch();
  const { user, loading, error } = useSelector((state) => state.user);

  useEffect(() => {
    dispatch(fetchUserData());
  }, [dispatch]);

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div>
      <h1>{user.name}</h1>
      <p>Email: {user.email}</p>
    </div>
  );
}

export default UserProfile;
```

# Redux Toolkit

– Современный и упрощенный вариант redux

# Redux Toolkit

— это набор инструментов для работы с Redux, который упрощает многие аспекты разработки.

Он помогает сократить количество шаблонного кода, предоставляет встроенную поддержку асинхронных действий и улучшает производительность.

RTK является официальной рекомендуемой библиотекой для работы с Redux.

## Основные функции Redux Toolkit:

1. `configureStore` — для создания хранилища с встроенными улучшениями.
2. `createSlice` — для упрощённого создания редьюсеров и экшенов.
3. `createAsyncThunk` — для работы с асинхронными запросами.
4. `createEntityAdapter` — для работы с нормализованными данными (например, списками объектов).
5. `Immer` — позволяет работать с неизменяемым состоянием (*immutable state*) в удобном виде.
6. Интеграция с `DevTools` и поддержка `redux-thunk` по умолчанию.

## 1. Установка Redux Toolkit

Для начала нужно установить Redux Toolkit и React-Redux (если ты работаешь с React):

```
npm install @reduxjs/toolkit react-redux
```

```
npm i @reduxjs/toolkit react-redux
```

# Redux Toolkit

— шаги по созданию и  
использования

## 2. Создание хранилища с configureStore

В классическом Redux Вы бы писали нечто вроде:

```
import { createStore, applyMiddleware } from 'redux'
import rootReducer from './reducers'
import thunk from 'redux-thunk'

const store = createStore(rootReducer, applyMiddleware(thunk))
export default store
```

А также отдельно настраивали бы DevTools через composeWithDevTools()

### В RTK

В Redux Toolkit есть функция configureStore, которая:

1. Автоматически подключает Redux Thunk в качестве middleware.
2. Автоматически включает поддержку Redux DevTools в режиме разработки.
3. Упрощает настройку дополнительных middleware.
4. Под капотом вызывает combineReducers, если вы передадите объект с редьюсерами.

```
// store.js
import { configureStore } from '@reduxjs/toolkit'
import someSlice from '../features/someFeature/someSlice'
import anotherSlice from '../features/anotherFeature/anotherSlice'

export const store = configureStore({
  reducer: {
    some: someSlice,
    another: anotherSlice,
  },
  // Опциональные настройки
  devTools: process.env.NODE_ENV !== 'production',
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: false, // например, если надо что-то отключить
    }),
})
```

# Redux Toolkit

— шаги по созданию и  
использования

## 3. `createSlice`: упрощённое создание редьюсера и экшенов

В классическом Redux Вы бы писали нечто вроде:

```
// actions.js
export const ADD_TODO = 'ADD_TODO'
export const REMOVE_TODO = 'REMOVE_TODO'

export function addTodo(payload) {
  return { type: ADD_TODO, payload }
}

export function removeTodo(payload) {
  return { type: REMOVE_TODO, payload }
}

// reducer.js
const initialState = {
  todos: []
}

function todoReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return { ...state, todos: [...state.todos, action.payload] }
    case REMOVE_TODO:
      return { ...state, todos: state.todos.filter(...) }
    default:
      return state
  }
}

export default todoReducer
```

В RTK: `createSlice` `createSlice` решает задачу одним махом:

1. Генерирует экшен-константы и экшен-криэйторы.
2. Создаёт редьюсер, где мы пишем «мутабельный» код (Immer обворачивает и обеспечивает неизменяемость).
3. Не нужны конструкции Switch и default case.

```
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  todos: []
}

const todosSlice = createSlice({
  name: 'todos',
  initialState,
  reducers: {
    addTodo(state, action) {
      // RTK и Immer позволяют «писать» в состояние,
      // фактически делая под капотом неизменяемые структуры
      state.todos.push(action.payload)
    },
    removeTodo(state, action) {
      state.todos = state.todos.filter(todo => todo.id !== action.payload)
    },
  },
})

export const { addTodo, removeTodo } = todosSlice.actions
export default todosSlice.reducer
```

`todosSlice.reducer` – это полноценный редьюсер,  
который мы подключаем в `configureStore`.

# Redux DevTools Extension

– Плагин для браузера для отладки redux приложений

# Redux DevTools Extention

— это плагин для браузера, который:

1. Позволяет просматривать текущее состояние хранилища (store).
2. Отслеживает действия (actions) и их порядок.
3. Позволяет «отмывать назад» (time travel debugging) и смотреть, как менялось состояние приложения со временем.
4. Упрощает поиск ошибок и анализ, как экшены влияют на состояние.

## Отдельный функционал от Redux или RTK

Redux DevTools Extension существует как отдельное расширение для браузеров и не включён непосредственно в код Redux или Redux Toolkit.

### Classic Redux с Redux DevTools Extention

Если по каким-то причинам вы не используете RTK, вы всё ещё можете отдельно подключить Redux DevTools, используя [redux-devtools-extension](#) пакет и enhancer при создании стора с помощью createStore:

```
import { createStore } from 'redux'
import rootReducer from './reducers'
import { composeWithDevTools } from 'redux-devtools-extension'

const store = createStore(
  rootReducer,
  composeWithDevTools()
)
```

И установите npm пакет `npm install @redux-devtools/extension`.

### RTK с Redux DevTools Extention

В Redux Toolkit есть встроенная интеграция с Redux DevTools через функцию configureStore. Если вы создаёте стор через configureStore, он по умолчанию умеет «общаться» с Redux DevTools (если расширение установлено и открыто в браузере)

```
import { configureStore } from '@reduxjs/toolkit'
import rootReducer from './reducers'

const store = configureStore({
  reducer: rootReducer,
  devTools: process.env.NODE_ENV !== 'production',
})
```

# Redux Toolkit

## **createAsyncThunk**

– Современное выполнение асинхронных действий

# Redux Toolkit

— шаги по созданию и  
использования

## 4. `createAsyncThunk`: упрощённая работа с асинхронными запросами

В обычном Redux, если вам нужно сделать запрос к серверу (или любую асинхронную операцию), вы бы:

1. Писали три экшена — REQUEST, SUCCESS, ERROR.
2. Писали «thunk» (функцию), которая вызывает `fetch` / `axios`, и внутри неё вручную диспатчили эти три экшена.

Это много шаблонного кода (boilerplate).

`createAsyncThunk` всё это автоматизирует и создаёт три экшена (`pending/fulfilled/rejected`) за вас.

### a) Объявляем асинхронную «задачу»

```
import { createAsyncThunk } from '@reduxjs/toolkit'

// 'users/fetchUsers' – это префикс для экшенов
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async () => {
    const response = await fetch('https://jsonplaceholder.typicode.com/users')
    // возвращаем то, что станет action.payload при fulfilled
    return await response.json()
  }
)
```

При вызове `dispatch(fetchUsers())`, Redux Toolkit автоматически создаёт три экшена:

1. `users/fetchUsers/pending` (начинаем запрос)
2. `users/fetchUsers/fulfilled` (запрос выполнился успешно, данные в `payload`)
3. `users/fetchUsers/rejected` (запрос «упал» с ошибкой)

Вы не пишете эти экшены вручную. `createAsyncThunk` сделает это за вас.

# Redux Toolkit

— шаги по созданию и  
использования

## b) Обрабатываем результат в слайсе

Чтобы отреагировать на эти экшены (например, изменить state.loading, state.error, положить данные в state.users), в Redux Toolkit используют extraReducers:

```
import { createSlice } from '@reduxjs/toolkit'
import { fetchUsers } from './thunks' // тот код, где мы объявили fetchUsers

const userSlice = createSlice({
  name: 'users',
  initialState: {
    users: [],
    loading: false,
    error: null,
  },
  reducers: {}, // обычные (синхронные) редьюсеры, если нужны
  extraReducers: (builder) => {
    // .pending → когда запрос начался
    builder.addCase(fetchUsers.pending, (state) => {
      state.loading = true
      state.error = null
    })
    // .fulfilled → когда запрос завершился успешно
    builder.addCase(fetchUsers.fulfilled, (state, action) => {
      state.loading = false
      state.users = action.payload // тут лежат данные с сервера
    })
    // .rejected → когда произошла ошибка
    builder.addCase(fetchUsers.rejected, (state, action) => {
      state.loading = false
      state.error = action.error.message // или action.payload, если используем rejectWithValue
    })
  },
})
export default userSlice.reducer
```

- Когда вы делаете dispatch(fetchUsers()), Redux Toolkit автоматически отправит users/fetchUsers/pending.
- В extraReducers поймается .pending и сделает state.loading = true.
- Когда запрос завершится, Toolkit отправит users/fetchUsers/fulfilled (если успех) или users/fetchUsers/rejected (если ошибка).
- В .fulfilled вы кладёте action.payload (данные с сервера) в state.users.
- В .rejected обрабатываете action.error или action.payload.

Вы не пишете вручную экшены REQUEST, SUCCESS, FAILURE — вся эта логика заложена в createAsyncThunk и extraReducers.

# Redux Toolkit

## — шаги по созданию и использования

### c) используем в компоненте React

Обычно это выглядит так:

```
import React, { useEffect } from 'react'
import { useDispatch, useSelector } from 'react-redux'
import { fetchUsers } from './thunks'

function UserList() {
  const dispatch = useDispatch()
  const { users, loading, error } = useSelector(state => state.users)

  useEffect(() => {
    dispatch(fetchUsers()) // запрос на сервер
  }, [dispatch])

  if (loading) return <p>Loading...</p>
  if (error) return <p style={{ color: 'red' }}>Error: {error}</p>

  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  )
}

export default UserList
```

1. При первом рендре useEffect вызывает dispatch(fetchUsers()).
2. fetchUsers() внутри себя вызывается и диспатчит users/fetchUsers/pending. Редьюсер ставит loading = true.
3. Когда ответ получен — диспатчится users/fetchUsers/fulfilled, редьюсер кладёт данные в state.users и loading = false.
4. Компонентrerендерится, показывая список пользователей.

# Обработка ошибок

## createAsynсThunk

– Современное выполнение асинхронных действий

# Redux Toolkit

— шаги по созданию и  
использования

## Есть 2 варианта обработки и получения ошибки

### 1 Вариант: «бросания» ошибки (throw new Error)

Если внутри вашего createAsyncThunk вы делаете:

```
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async (arg, { getState, dispatch }) => {
    const response = await fetch('https://jsonplaceholder.typicode.com/users')

    // Проверяем, нормальный ли статус ответа
    if (!response.ok) {
      // Если статус не ок, то просто бросаем ошибку
      throw new Error(`Ошибка! Статус ${response.status}`)
    }

    const data = await response.json()
    return data
  }
)
```

Тогда в вашем extraReducers вы указываете:

```
extraReducers: (builder) => {
  builder
    .addCase(fetchUsers.pending, (state) => {
      state.loading = true
      state.error = null
    })
    .addCase(fetchUsers.fulfilled, (state, action) => {
      state.loading = false
      state.users = action.payload
    })
    .addCase(fetchUsers.rejected, (state, action) => {
      state.loading = false
      // Если мы «бросили» ошибку, она окажется в action.error.message
      state.error = action.error.message
    })
}
```

В этом случае, если throw new Error(...) сработает, Redux Toolkit «поймет» эту ошибку и задиспатчит экшен fetchUsers.rejected с полем action.error.message = 'Текст ошибки'.

# Redux Toolkit

— шаги по созданию и  
использования

## Есть 2 варианта обработки и получения ошибки

### 2. Вариант с rejectWithValue (специальная обработка)

Иногда хочется передать какие-то дополнительные данные в ошибку, а не только message. Тогда используем rejectWithValue.

```
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async (_, { rejectWithValue }) => {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/users')
      if (!response.ok) {
        // Здесь мы не бросаем ошибку, а возвращаем «обработанный» текст или объект
        return rejectWithValue(`Ошибка при загрузке. Статус: ${response.status}`)
      }
      const data = await response.json()
      return data
    } catch (err) {
      // Сетевая ошибка, например
      return rejectWithValue(err.message || 'Неизвестная ошибка')
    }
  }
)
```

Тогда в вашем extraReducers:

```
extraReducers: (builder) => {
  builder
    .addCase(fetchUsers.pending, (state) => {
      state.loading = true
      state.error = null
    })
    .addCase(fetchUsers.fulfilled, (state, action) => {
      state.loading = false
      state.users = action.payload
    })
    .addCase(fetchUsers.rejected, (state, action) => {
      state.loading = false
      /*
        Если используем rejectWithValue → текст ошибки будет в action.payload.
        Иначе, если бросали ошибку → смотрим в action.error.message.
      */
      if (action.payload) {
        // Обработанная «своя» ошибка
        state.error = action.payload
      } else {
        // Брошенная ошибка (throw new Error)
        state.error = action.error.message
      }
    })
}
```

Таким образом, используя rejectWithValue, вы решаете, что именно хотите положить в поле action.payload при ошибке.

А потом в rejected проверяете:

- если action.payload существует — значит, это ошибка из rejectWithValue;
- иначе смотрите action.error.message (брошенная ошибка).

дополнительные

возможности

**createAsynсThunk**

– Современное выполнение асинхронных действий

# Redux Toolkit

— шаги по созданию и  
использования

## Общий вид функции в createAsyncThunk

В колбэке, который вы передаёте во второй аргумент createAsyncThunk, может быть два параметра:

```
export const someThunk = createAsyncThunk(
  'something/someThunk',
  async (arg, thunkAPI) => {
    // ...
  }
)
```

первый параметр (здесь назван arg) — это значение, которое вы передаёте, когда вызываете экшен:

```
dispatch(someThunk(42))
// тогда внутри будет arg = 42
```

второй параметр (здесь назван thunkAPI) — это объект, который даёт доступ к дополнительным методам Redux Toolkit. Самые нужные:

- dispatch — чтобы диспатчить другие экшены внутри thunk.
- getState — чтобы посмотреть текущий Redux-стейт.
- rejectWithValue — чтобы при ошибке вернуть «особое» значение в action.payload.

## Почему часто пишут `(_, { rejectWithValue })`

«`_`» вместо «`arg`»

Если не используете первый параметр (аргумент), его часто называют `_` или `__`, чтобы было понятно: «я его не использую». Это не специальное слово, а просто соглашение. JavaScript позволяет вам назвать параметр хоть `_`, хоть `banana`.

`{ rejectWithValue }` вместо `thunkAPI`

Если вам из `thunkAPI` нужен только метод `rejectWithValue`, то программисты делают деструктуризацию:

```
async (_, { rejectWithValue }) => {
  // ...
}
```

# Redux Toolkit

— шаги по созданию и  
использования

## Второй параметр (здесь назван thunkAPI)

Когда вы пишете колбэк для `createAsyncThunk`, второй параметр называется условно `thunkAPI`. Он даёт доступ к нескольким методам: `dispatch`, `getState`, `rejectWithValue`, `fulfillWithValue` и т.д.

```
export const myAsyncThunk = createAsyncThunk(
  'someSlice/myAsyncThunk',
  async (arg, thunkAPI) => {
    // Здесь вы можете:
    // 1. Диспатчить любые экшены:
    thunkAPI.dispatch(someAction())

    // 2. Получить текущий Redux state
    const state = thunkAPI.getState()
    console.log('Current state:', state)

    // Пример асинхронного запроса
    const response = await fetch('https://jsonplaceholder.typicode.com/users')
    const data = await response.json()

    // Возвращаем данные
    return data
  }
)
```

### Обратите внимание:

1. `arg` — это то, что вы передаёте в `dispatch(myAsyncThunk(argValue))`.
2. `thunkAPI` — содержит методы:
  - `dispatch`: позволяет диспатчить другие экшены внутри этого thunk.
  - `getState`: возвращает текущий Redux-стейт.
  - `rejectWithValue`: для передачи «специальной ошибки» в `action.payload` при `rejected`.
  - `fulfillWithValue`: более редкий случай, аналог для «успешного» возврата.
  - `requestId`, `signal`, `abort`: продвинутые штуки для отмены запроса или привязки к `AbortController`.

```
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async (arg, { dispatch, getState, rejectWithValue }) => {
    // Для примера: проверим что-то в стейте
    const { users } = getState().userData
    if (users.length > 0) {
      // уже есть пользователи → не загружаем повторно
      return users
    }

    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/users')
      if (!response.ok) {
        // Можем сделать дополнительный диспач тут:
        dispatch(showNotification('Ошибка при загрузке пользователей'))
        return rejectWithValue(`Статус: ${response.status}`)
      }
      const data = await response.json()
      return data
    } catch (error) {
      dispatch(logError(error)) // Логирование, например
      return rejectWithValue(error.message || 'Неизвестная ошибка')
    }
  }
)
```

# Redux Toolkit

— шаги по созданию и  
использования

## 5. Работа с нормализованными данными с `createEntityAdapter`

`createEntityAdapter` полезен для работы с коллекциями данных (например, списками объектов), обеспечивая лёгкое добавление, обновление и удаление элементов. Он автоматически нормализует данные и управляет коллекциями через стандартные методы.

```
import { createSlice, createEntityAdapter } from '@reduxjs/toolkit';
const usersAdapter = createEntityAdapter();
// Начальное состояние с адаптером
const initialState = usersAdapter.getInitialState();
const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    addUser: usersAdapter.addOne, // Добавляем одного пользователя
    updateUser: usersAdapter.updateOne, // Обновляем одного пользователя
    removeUser: usersAdapter.removeOne, // Удаляем пользователя
  },
});
export const { addUser, updateUser, removeUser } = usersSlice.actions;
export default usersSlice.reducer;
```

- Адаптеры создают удобные методы для работы с коллекциями данных.
- Эти методы сокращают количество кода и делают его более выразительным.

## 6. Подключение хранилища к React через `Provider`

Как и в обычном Redux, для использования хранилища в React-приложении нужно подключить его через компонент `Provider`.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

## 7. Интеграция с `DevTools`

`configureStore` по умолчанию включает поддержку Redux DevTools, поэтому тебе не нужно делать дополнительную настройку для их использования в разработке.

Если ты хочешь отключить `DevTools` в продакшене, можешь сделать это так:

```
const store = configureStore({
  reducer: rootReducer,
  devTools: process.env.NODE_ENV !== 'production', // Включаем только в режиме разработки
});
```

# UseMemo

– Оптимизация работы приложения

# useMemo()

— это хук, который мемоизирует результат вычисления функции. Это означает, что если зависимости не изменяются, useMemo возвращает ранее вычисленное значение, вместо того чтобы выполнять вычисления заново при каждом рендрере.

— Можете скопировать код из комментариев ниже и протестировать

## Отличие от memo()

Протестируйте код из комментариев.

1-Увеличьте count - нажатием кнопки

2-Покажите статьи - нажатием кнопки

3-Увеличьте count - нажатием кнопки

Увидите что проблема осталась. Хотя мы использовали обертку memo()

### Проблема осталась,

так как в этот раз мы использовали в качестве props объект.

А как мы знаем, если, например, сравнить два объекта с одинаковыми данными, то мы получим false.

Объекты не будут равны, так как они сравниваются по ссылке, а не по значению.

Пример: `({}) != {}`

### Почему в данном случае не работает memo()?

Функция memo() предотвращает повторный рендер компонента, если его пропсы не изменились. Однако в примере это не срабатывает, потому что передаётся объект archiveOptions в качестве пропса.

Объекты в JavaScript сравниваются по ссылке, а не по значению.

Это означает, что каждый раз, когда компонент App рендерится, создается новый объект archiveOptions, даже если его содержимое остается неизменным.

В таком случае memo() думает, что пропсы изменились, потому что сравнивает ссылки на объекты (а не их содержимое), и поэтому компонент Archive будет перерендеряться каждый раз, когда перендерится компонент App.

# useMemo()

— Для того чтобы избежать создания нового объекта на каждом рендере, можно использовать useMemo.

Он мемоизирует результат вычисления и сохраняет объект, пока его зависимости не изменились.

## Отличие от memo()

Протестируйте код из комментариев.

- 1-Увеличьте count - нажатием кнопки
- 2-Покажите статьи - нажатием кнопки
- 3-Увеличьте count - нажатием кнопки

Увидите что проблема решена. Хотя мы и используем объект в качестве props

```
function App() {
  const [count, setCount] = useState(0);

  // Используем useMemo для мемоизации archiveOptions
  const archiveOptions = useMemo(
    () => {
      show: false,
      title: "Archived Posts",
    },
    []
  );

  return (
    <div>
      <h1>Posts</h1>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setShow(!show)}>Show</button>
      <ul>
        {archiveOptions.map((option) => (
          <li>{option.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

### Как работает useMemo() в этом случае?

Для того чтобы избежать создания нового объекта на каждом рендере, можно использовать useMemo. Он мемоизирует результат вычисления и сохраняет объект, пока его зависимости не изменились. В данном случае, вы можете мемоизировать archiveOptions, чтобы объект не создавался заново при каждом рендере.

# useMemo()

— Подробнее

## useMemo()

— это хук, который мемоизирует результат функции и возвращает его только тогда, когда изменяются зависимости. Его синтаксис выглядит следующим образом:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- **Первый аргумент:** Это функция, которая возвращает значение.

Это может быть вычисление, которое занимает много времени или ресурсов.

- **Второй аргумент:** Массив зависимостей. Этот массив говорит useMemo, когда нужно пересчитать значение.

Если ни одна зависимость не изменилась, то useMemo вернет ранее мемоизированное (сохраненное) значение.

## Синтаксис useMemo()

```
const memoizedValue = useMemo(() => {
  // функция, которая возвращает результат
  return сложные_вычисления(a, b);
}, [a, b]); // массив зависимостей
```

## Как работает массив зависимостей?

Массив зависимостей — это список переменных, от которых зависит результат функции в useMemo. Если значения переменных в массиве изменяются прирендере компонента, useMemo пересчитает мемоизированное значение, иначе вернет ранее вычисленный результат.

1. Если зависимость изменяется: Функция передается в useMemo и будет выполнена заново, и новое значение будет возвращено и запомнено.
2. Если зависимость не изменяется: useMemo просто вернет ранее мемоизированное значение и не будет вызывать функцию заново.

**Пустой массив []:** Если вы передаете пустой массив зависимостей, то функция, переданная в useMemo, выполнится только один раз — при монтировании компонента. Это похоже на поведение useEffect с пустым массивом зависимостей:

В этом случае функция выполнится только один раз при монтировании, и результат будет использован до тех пор, пока компонент не будет удален. **ПРОТЕСТИРУЙТЕ КОД ИЗ КОММЕНТАРИЯ**

# UseCallback

– Оптимизация работы приложения

# useCallback()

— это хук React, который возвращает мемоизированную версию функции, которая сохраняет одну и ту же ссылку на функцию между рендерами, если зависимости (указанные в массиве зависимостей) не изменились.

Это полезно для предотвращения ненужного пересоздания функций на каждом рендрере.

— Можете скопировать код из комментариев ниже и протестировать

## Как работает useCallback?

Когда компонент рендерится, каждый раз создаются новые версии функций, передаваемых в дочерние компоненты, и если функции передаются через пропсы, дочерний компонент будет перерендеряться, даже если его пропсы фактически не изменились.

useCallback решает эту проблему, мемоизируя функцию и предотвращая её пересоздание, если зависимости не изменяются.

### Синтаксис useCallback:

```
const memoizedCallback = useCallback(() => {  
  // логика функции  
}, [dependencies]);
```

- Первый аргумент — это функция, которая будет мемоизироваться.
- Второй аргумент — это массив зависимостей. Если какие-либо зависимости изменяются, функция пересоздается.

### ПРОТЕСТИРУЙТЕ 2 ВАРИАНТА КОДА ИЗ КОММЕНТАРИЯ

#### Пример 1: Без useCallback (Проблема с производительностью)

В этом примере каждый раз, когда состояние компонента изменяется, пересоздается функция, что приводит к перендеру дочернего компонента, даже если его пропсы не изменились.|

##### Проблема:

- Каждый раз, когда вы нажимаете кнопку “Increase Count”, рендерится компонент App, и функция handleClick пересоздается.
- Это вызывает ненужный перендер компонента PostList, несмотря на то, что его пропсы не изменились, что ухудшает производительность при большом количестве постов.

#### Пример 2: С useCallback

- Теперь функция handleClick мемоизируется с помощью useCallback. Она создается один раз при первом рендрере компонента App и не пересоздается при изменении состояния count.
- Это предотвращает ненужные перендеры компонента PostList, так как его пропсы больше не меняются при каждом рендрере.

# useCallback()

— Подробнее

## useCallback()

— это хук в React, который используется для мемоизации функций.

Он возвращает ту же самую функцию, если её зависимости не изменяются. Это полезно для предотвращения ненужного пересоздания функций при каждом рендере, особенно если функция передается как пропс в дочерний компонент, что может вызвать лишниеrerендеры.

## Когда использовать useCallback?

- Когда функция зависит от переменных (например, состояния или пропсов) и вы хотите контролировать, когда она должна пересоздаваться.
- Когда у вас есть сложная или ресурсоёмкая функция, которая не должна пересоздаваться на каждом рендере компонента.

```
const memoizedCallback = useCallback(() => {  
    // Логика функции  
, [dependency1, dependency2]);
```

- Первый аргумент — это функция, которую вы хотите мемоизировать.
- Второй аргумент — это массив зависимостей. Если хотя бы одна из зависимостей изменяется, useCallback создаёт новую версию функции.

## Как работает массив зависимостей?

Массив зависимостей в useCallback указывает, от каких значений зависит функция. Функция пересоздается только в случае изменения одной из зависимостей в этом массиве. Если зависимости остаются неизменными, useCallback вернёт ту же самую функцию, которая была создана ранее.

# Bundle & LazyLoad

– Оптимизация работы приложения

# Bundle Splitting

— это техника оптимизации, при которой большой JavaScript-бандл разбивается на несколько более мелких частей.

Это позволяет загружать только те части приложения, которые необходимы для текущего отображения, вместо того чтобы загружать весь код сразу.

## Без разбиения бандлов:

Приложение загружает весь код сразу, даже если некоторые компоненты не будут использоваться немедленно.

## С разбиением бандлов:

Приложение загружает только то, что необходимо для текущего экрана, а оставшийся код загружается только при необходимости (например, когда пользователь переходит на другую страницу или взаимодействует с определённой частью приложения).



# Lazy Loading

— это техника, которая позволяет загружать код компонента только тогда, когда он действительно необходим (например, когда компонент должен быть отображён на экране)

## Lazy Loading в React

React поддерживает ленивую загрузку компонентов с помощью встроенного хука `React.lazy()`.

```
import React, { Suspense } from 'react';

// Ленивый импорт компонента
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <h1>Welcome to the app</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent /> /* Этот компонент будет загружен только при его рендрере */
      </Suspense>
    </div>
  );
}

export default App;
```

В этом примере компонент `LazyComponent` загружается только тогда, когда он должен быть отрендерен.

`Suspense` — это компонент, который позволяет указать запасной контент (например, “`Loading...`”), который будет показан, пока ленивый компонент загружается.

## Как связаны Bundle Splitting и Lazy Loading?

Оба эти метода работают вместе, чтобы оптимизировать загрузку JavaScript-бандлов в React-приложении:

- `Bundle splitting` позволяет разделить код на более мелкие части (бандлы), каждый из которых может быть загружен отдельно.
- `Lazy loading` использует эти разделённые бандлы, чтобы загружать их только тогда, когда они необходимы.

Ленивую загрузку можно применять как к страницам так и к компонентам

```
import React, { Suspense } from 'react';

// Лениво загружаем компонент
const MyComponent = React.lazy(() => import('./MyComponent'));

function App() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <MyComponent /> /* Этот компонент будет загружен только когда понадобится */
      </Suspense>
    </div>
  );
}

export default App;
```

# Хэширование

## изображений

– Подготовка изображений к деплою

# Изображения

— Размещайте изображения  
через `import` в тот или иной  
компонент

## Хэширование изображений

Рекомендуется в Vite (а также в большинстве современных сборщиков) делать импорт картинки и использовать её как модуль, чтобы Vite смог оптимизировать/хешировать её

```
import logo from '../assets/images/logo.svg';

function Header() {
  return (
    <header className="flex justify-between px-5 py-8 bg-blue-200 shadow">
      <img className="h-6" src={logo} alt="logo" />
      {/* ...остальное... */}
    </header>
  );
}

export default Header;
```

Когда мы говорим, что бандлер (Vite, Webpack и т.д.) «хеширует» файлы, имеется в виду, что в итоговом бандле (папке `dist`) у файлов (картинок, JS, CSS и пр.) появляются сгенерированные «дополнения» в имени — обычно это набор букв и цифр, например:

```
logo.svg → logo.abcd1234.svg
```

Этот «хэш» (часто MD5 или другой алгоритм) меняется, когда сам файл меняется, и служит двум главным целям:

**1. Уникальность имени.** Если пользователь (или прокси-сервер) закешировал файл под старым именем, то при следующем билде, когда содержимое файла изменилось, у него уже будет другое имя (с другим хэшом). Благодаря этому браузер понимает, что файл новый, и загружает его заново (а не достаёт из кэша).

**2. Борьба со «сломанными» кэшами.** Без хэша браузеры могут не замечать изменения, например, если у вас включён агрессивный кэш. Тогда пользователи не увидят новые версии сразу. Когда есть хэш в названии, любое изменение в файле гарантированно даёт новое имя.

# NPM Run Build

– Сборка проекта Vite

# NPM Run Build

— Команда `npm run build` в проекте на основе Vite и React используется для создания оптимизированной версии вашего приложения, готовой к развертыванию на продакшн-сервере.

## Что делает команда `npm run build`:

### Создаёт оптимизированный бандл:

- Vite собирает все ресурсы вашего приложения (HTML, CSS, JavaScript, изображения) и объединяет их в минимизированный и оптимизированный для продакшена бандл.
- Бандл включает только те части кода, которые используются в приложении (tree-shaking).

### Минифицирует код:

- JavaScript и CSS минимизируются (удаляются пробелы, комментарии и лишние символы), чтобы уменьшить размер файлов.
- Используется библиотека [esbuild](#) для быстрой минификации.

### Оптимизирует загрузку:

- Генерирует ссылки для ленивой загрузки (dynamic imports) в ваших модулях.
- Асинхронные модули загружаются только тогда, когда они необходимы.

### Создаёт папку dist:

- Все собранные и оптимизированные файлы складываются в папку dist, которая готова для загрузки на сервер.

### Оптимизирует статические ресурсы:

- Изображения (например, .png, .jpg, .svg) могут быть оптимизированы, если вы используете соответствующие плагины (например, vite-imagemin).
- Файлы из папки public/ копируются в папку dist без изменений.

# Деплой проекта

– Размещение сайта на хостинге

# Запустите скрипт и разместите файлы на хостинге

— Размещение файлов проекта  
на сервер вручную

## “Локальная сборка + ручная загрузка”

1. Локально делаете npm run build
2. Берёте содержимое папки dist/.
3. Заходите в панель управления хостингом (например, ISPmanager, cPanel или что-то ещё), через встроенный файловый менеджер или SFTP/FTP загружаете все файлы из dist/ в корневую директорию сайта (или в нужный подкаталог).
4. Настраиваете, чтобы домен указывал именно на эту папку (если нужно).