React JS

— React JS Course

REACT JS

Navigation

REACT JS

Navigation

Essential Theory

45. Class vs Functional Components
46. Components, instances, elements
47. How Rendering Works
48. What is the Virtual DOM
49. Fiber tree
50. Fiber Tree vs Virtual DOM
51. Rendering Flowchart
52. Component Memoization
53. Commit Phase
54. Diffing
55. State Updates Batching
56. Mounting and Unmounting
57. Framework vs library

useRef & Custom Hook

58. useRef Hook
59. Custom Hook

React Router

60. React Router Library
61. <Link> Component
62. <Outlet /> Component
63. index Attribute
64. useParams() hook
65. useSearchParams() hook
66. useLocation() hook
67. useNavigate() hook
68. Компонет <Navigate />
69. Ways to Create a Link

CSS Modules

70. CSS Modules

React Router 6.4+ (with Data Loading)

71. Router before v6.4
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

Navigation

Redux + RTK

79. Redux Introduction
80. Redux Thunk
81. Redux Toolkit (RTK)
82. Redux DevTools Extention
83. Toolkit createAsyncThunk
84. Error Handling (RTK)
85. Additional Features of AsyncThunk

Performance Optimization

86. UseMemo hook
87. UseCallback hook
88. Boundle **&** LazyLoad

Project Deployment

89. Image Hashing
90. NPM Run Build
91. Project Hosting

Your **Questions**

— and How to Ask Them

I Answer Questions Every Day

I answer questions every day only on the platform where you are taking the course.

Please do not write to me on Instagram or Telegram

You can ask questions in the comments under the video or in the platform's chat

— depending on what the platform provides.

Detailed Question

Please ask your questions as thoroughly and with as much detail as possible.

- Do not paste your code directly into the message text.
- Do not ask questions like “It doesn’t work, what should I do?” without detailed explanations of your actions and what exactly is not working.
- Do not write “I don’t understand” without explaining in detail what exactly you don’t understand.
- Do not send cropped screenshots. (I will zoom in on the needed area)
- Do not ask long, confusing, or philosophical questions. A couple of lines is always enough.

Do not paste your code directly into the message text.

Please get familiar with GitHub or at least Google Drive to provide me with a link to your entire project.

Don’t forget to remove unnecessary elements from your project — for example, the node_modules folder, which can take up more than 200 MB.

clip-patch можно ли устоновить иконку svg за приделами изображения clip-patch? чтобы иконку было видно. Спасибо

например иконку сейчас вижу

```
видно иконку

*, ::after, ::before

    margin: 0
    padding: 0
    box-sizing: border-box


body


    font-famyli: "Roboto", sans-serif
    font-weight: 400
    color: #777777
    font-size: 1.6rem
    letter-spasing: .2rem
    padding: 5rem
    background-color: white


.container

    background-color: #f7f7f7


.header


    position: relative
    height: 130vh
    background: linear-gradient(90deg, rgba(186, 133, 84, 0.7) 0.03%, rgba(1, 1, 1, 0.7) 99.94%), url('../img/bmw1.jpg') center / cover no-repeat
    clip-path: polygon(50% 0%, 83% 12%, 100% 43%, 94% 78%, 68% 100%, 32% 100%, 6% 78%, 0% 43%, 17% 12%)
```

REACT JS

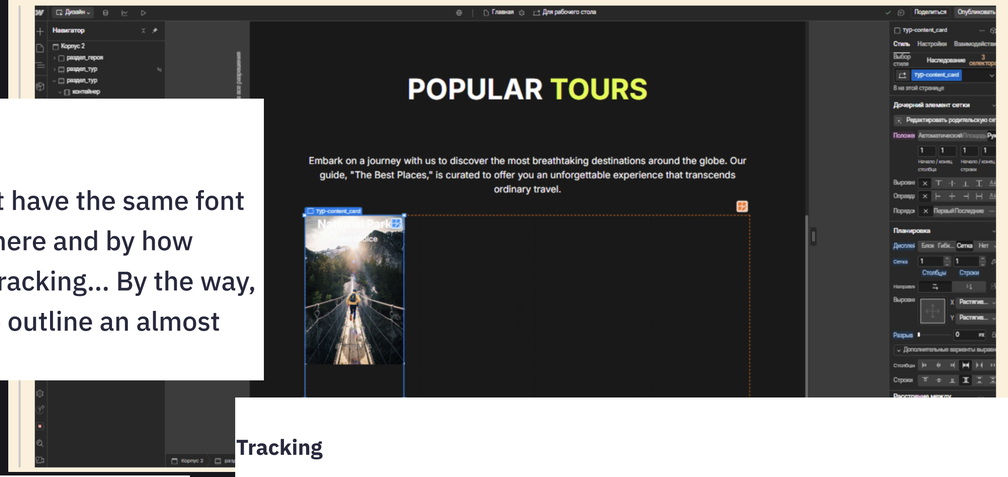
Do not ask long or philosophical questions.

I did everything as you said, but when I deleted 3 cards and replaced them with the first redesigned one, here's what happens — Webflow refuses to show them even though they exist — it's like they're here and not here at the same time. Please advise how to fix this, I can't move forward with the course because of it. I think Webflow just can't handle it — it's as if everything gets "broken" when I copy and paste as you showed. I had to roll back and start over several times at the exact moment of stress. I suspect that Webflow simply can't deal with a bunch of div blocks and grids nested into each other too many times, too many small glitches, and the perfect interface and tool performance itself suffers. I noticed that sometimes when trying to click on something — like buttons, for example — Webflow replaced my button with a div block and nothing helps fix the border. Even reassigning inheritance and visual editing of the same element in Webflow doesn't fix it without some kind of glitches. The hidden (closed) settings interface also started to glitch when I began creating blocks inside a bunch of grids (in the screenshot). You start building from scratch and nonsense comes out again, even though I follow your video exactly — it all shifts within a second and gets messed up.

P.S. Don't mind that everything is in Russian — I accidentally clicked "translate page" in Yandex, it's not intentional.

In short, Webflow lives its own life and doesn't want to deal with a bunch of grids and div blocks, it just wants the text to be a picture, already with text overlaid, because once the text is pinned, everything else breaks and shifts. Ideally, users should only be allowed to use containers and blocks with predefined parameters and a limited amount of functionality. Webflow now looks like an outdated tool from 9–12 years ago, and if this continues, the site will end up visually messy. You'll constantly get annoyed, things won't line up, you won't understand why the video results are clear but yours aren't. That's a question not for the teacher, but for Webflow itself (the development team).

I really hope you re-record this course at least with inserts, because the interface is outdated, though I understand this might be hard to manage.



Tracking

It's interesting that the text above the heading (the one that had to be set in uppercase) and the paragraph text have the same font size, but the first one needed to be stretched by 5%. It's clear that with practice comes an understanding of where and by how much this tracking should be applied. If only it were possible to somehow "formulate" a recommendation for tracking... By the way, regarding line spacing, in practice the adjustments across several points (font size; line spacing) allowed me to outline an almost linear graph of how one depends on the other — which is really helpful.

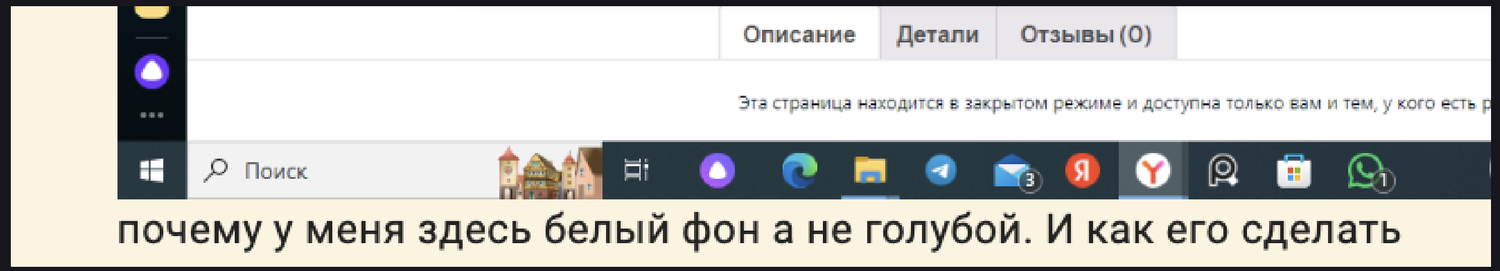
Actually, my question is related to the one user Anna asked 5 months ago. Around the 7th minute of the video, you create the first purple square of arbitrary size with a margin of 4 px, and then, selecting the square and the text inside, you apply Auto Layout with the required paddings. In your video, this automatically causes the square to shrink. For me, it didn't happen. Comparing the video with my screen, I realized that somehow the frame's Horizontal/Vertical resizing setting was switched to Hug instead of Fixed width, which was shown on the blue bar under the frame (previously there were specific dimensions, now it's Hug*Hug). But when and why this happened is unclear. I need clarification.

Tracking

It's interesting that the text above the heading (the one that had to be set in uppercase) and the paragraph text have the same font size, but the first one needed to be stretched by 5%. It's clear that with practice comes an understanding of where and by how much this tracking should be applied. If only it were possible to somehow "formulate" a recommendation for tracking... By the way, regarding line spacing, in practice the adjustments across several points (font size; line spacing) allowed me to outline an almost linear graph of how one depends on the other — which is really helpful.

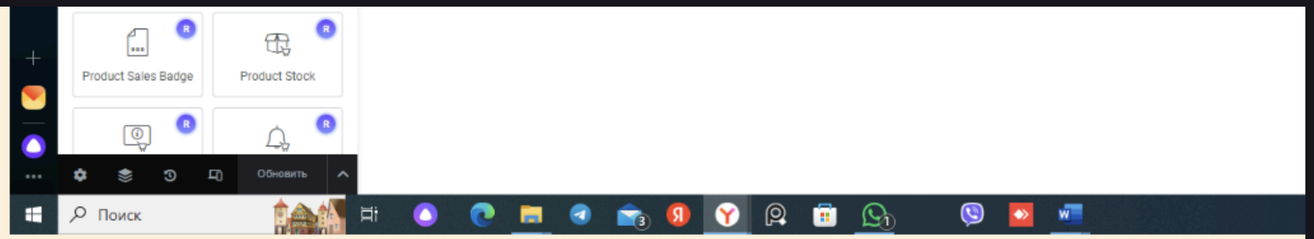
Dmitrii Fokeev:
Hi! Could you rephrase your question into 1–2 sentences, please? What's the problem with the structure? What exactly isn't working?

Do not ask questions like “It doesn’t work, what should I do?”



I can only help if you send a detailed and clear question, along with supporting materials: screenshots, a screen recording, or a link to your project with the code.

Do not ask questions like “I did everything as in the video, but it doesn’t work.”



сделала все как в этом видел никакие картинке в разделе товар не появились

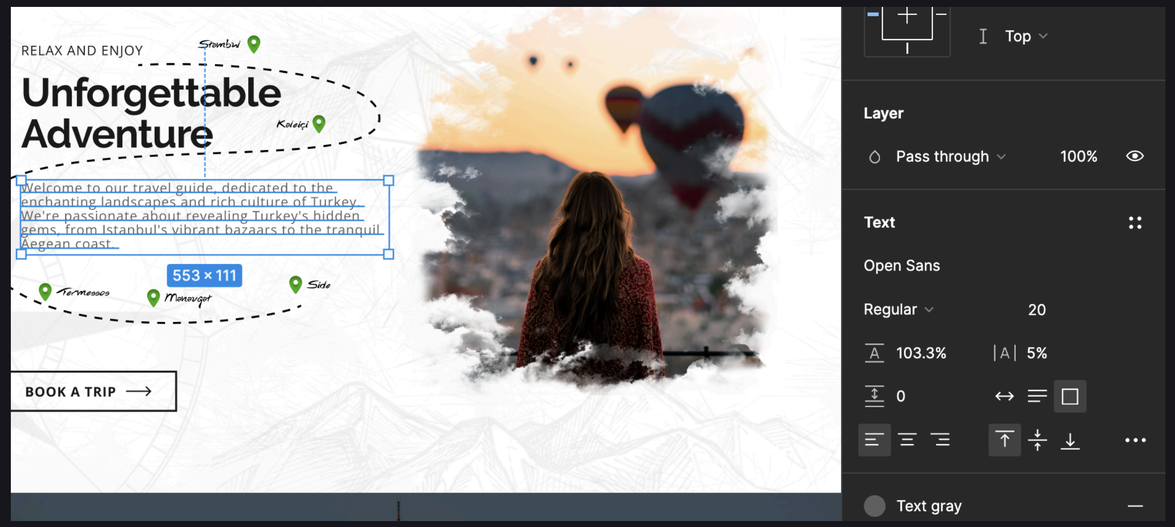
1. Check the comments under the video.

2. Make a second and third attempt, repeating everything slowly and exactly as shown in the video. In 99.9% of cases, the reason is a typo or mistake.

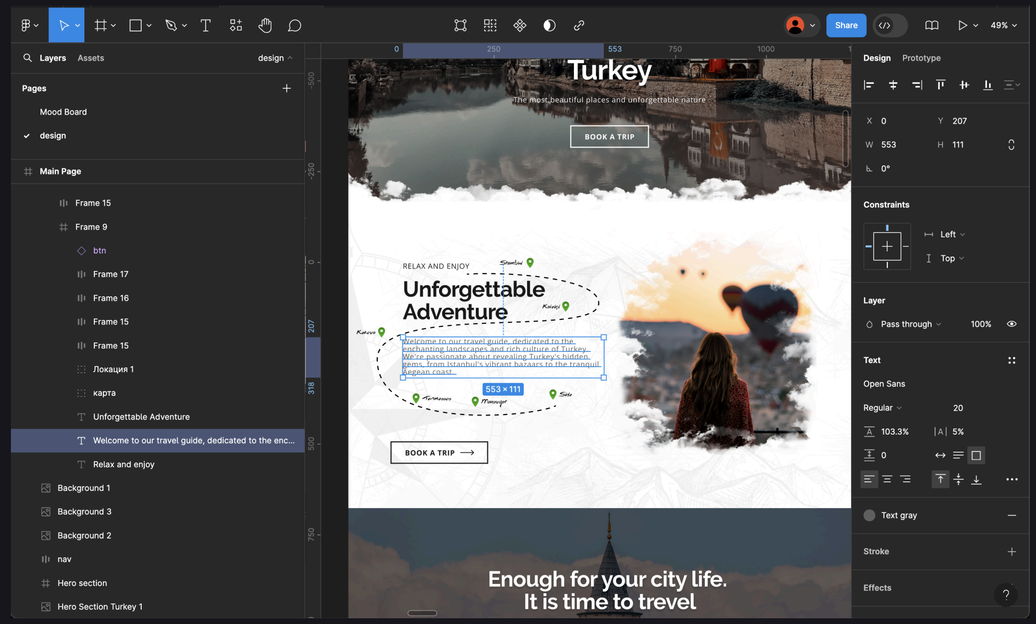
If something works in my video, it should work for you as well.

Do not send cropped screenshots. (I will zoom in on the needed area.)

not like that



like that



Compare your code with mine.

In the course materials, you always have access to all the code we go through in the lessons, as well as the complete code from the assignments. If you encounter errors, compare your code with the provided one.

How to Take the Course

— 3 options

Video Course

— The entire course consists of short video lessons in a “watch and repeat” format.

The course **is alive**

If you are taking the course on one of the official platforms, it means that the author moderates the course daily: all comments and suggestions are taken into account and corrected, so the large student community influences the quality of the course every day.

The course is **yours forever**

Access to the course is not limited by time.

The course **is updated**

If the material becomes outdated, the author updates it whenever possible. The course is always fresh.

No homework — only practice

The course includes many practical tasks to reinforce the material. It’s up to you whether to do them or not. You are already adults. The same applies to tests. But if you want to share your results, it’s always welcome.

Handbook

— A special interactive handbook has been created for the course with all the theory, guides, and rules.

— Для использования всех возможностей методички необходима программа Figma, которую вы можете скачать с официального сайта.

Quick Search

Use Ctrl / Cmd + F

Copy code from comments

In the comments for some lessons, there is code you can copy and experiment with.

Printable version in the course materials

A black-and-white version available in the course materials.

EASY AND FAST RESULT

2–3 months of learning

MEDIUM DIFFICULTY, MORE TIME

3–4 months of learning

DIFFICULT, SOMETIMES TEDIOUS AND LONG

4–5 months of learning

Overview of React capabilities

— An overview of React’s features and capabilities with simple examples.
I show and explain — you repeat the code.
(It’s best to do it several times and after a break)

Additional practice and tests

— Everything from the previous point + practical tasks and tests to reinforce the material.

Project creation

— Everything from the previous two points + project development.

VS Code

— Keyboard Shortcuts

Hotkeys for Mac:

Visual Studio Code

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
⇧⌘\	Jump to matching bracket
⌘] / ⌘[	Indent/outdent line
Home / End	Go to beginning/end of line
⌘↑ / ⌘↓	Go to beginning/end of file
^PgUp / ^PgDn	Scroll line up/down
⌘PgUp /⌘PgDn	Scroll page up/down
⌥⌘[/ ⌥⌘]	Fold/unfold region
⌘K ⌘[/ ⌘K ⌘]	Fold/unfold all subregions
⌘K ⌘0 / ⌘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
⇧⌥I	Insert cursor at end of each line selected
⌘L	Select current line
⇧⌘L	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
⌘\	Split editor
⌘1 / ⌘2 / ⌘3	Focus into 1st, 2nd, 3rd editor group
⌘K ⌘← / ⌘K ⌘→	Focus into previous/next editor group
⌘K ⇧⌘← / ⌘K ⇧⌘→	Move editor left/right
⌘K ← / ⌘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
⌘K Z	Zen Mode (Esc Esc to exit)

Debug

F9	Toggle breakpoint
F5	Start/Continue
F11 / ⇧F11	Step into/ out
F10	Step over
⇧F5	Stop
⌘K ⌘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

Hotkeys for Windows:

Visual Studio Code

Keyboard shortcuts for Windows

General

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

Basic editing

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

Navigation

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

Search and replace

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

Multi-cursor and selection

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

Rich languages editing

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

Editor management

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

File management

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

Display

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

Debug

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

Integrated terminal

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

Other operating systems' keyboard shortcuts and additional unassigned shortcuts available at aka.ms/vscodekeybindings

Terminal **Basics**

— Basic Commands and Hotkeys

If You’re Using macOS

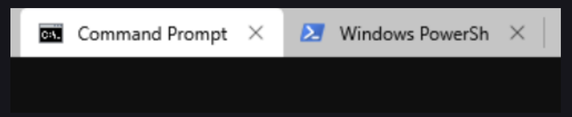
You have a single Terminal application regardless of your OS version.
Everything is fine — there’s nothing you need to figure out.

If You’re Using Windows

You’re using one of the three possible terminal programs:

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

You can identify which terminal you’re using by looking at the name of the terminal window after launching it (the terminal in VS Code is also labeled accordingly).



TERMINAL COMMANDS

macOS Terminal (Terminal.app) и Linux Terminal:

Folders and Paths:

1. Viewing Directory Contents:
• ls: Displays a list of files and folders.

2. Changing Directory:
• cd /Users/Username/Documents: Navigates to the specified directory.

3. Copying Files:
• cp file.txt /Volumes/Backup/file.txt: Copies a file to the specified location.

4. Deleting Files:
• rm file.txt: Deletes the specified file.

5. Moving Files:
• mv file.txt /Volumes/Backup/: Moves the file to the specified location.

6. Creating a Folder:
• mkdir NewFolder: Creates a new directory.

7. Deleting a Folder:
• rmdir NewFolder: Deletes an empty folder.

8. Creating a File:
• touch newfile.txt: Creates an empty file.

9. Clearing the Screen:
• clear: Clears the terminal screen.

10. Exiting the Terminal:
• exit: Ends the terminal session.

Hotkeys:

1. Clear Screen: Ctrl + L

2. Command History: Use the up/down arrow keys to scroll through previous commands.

3. Interrupt Command: Ctrl + C

4. Auto-Complete: Tab

5. Search Command History: Ctrl + R

6. Move Cursor by Word: Alt + ← / →

7. Delete Word Before Cursor: Ctrl + W

8. Delete Line Before Cursor: Ctrl + U

TERMINAL COMMANDS

If You're Using Command Prompt (cmd.exe) on Windows:

Folders and Paths:

1. View Directory Contents:
• dir: Displays a list of files and folders in the current directory.

2. Change Directory:
• cd C:\Users\Username\Documents: Moves to the specified directory.

3. Copy Files:
• copy file.txt D:\Backup\file.txt: Copies the file to the specified location.

4. Delete Files:
• del file.txt: Deletes the file.

5. Move Files:
• move file.txt D:\Backup\: Moves the file.

6. Create Folder:
• mkdir NewFolder: Creates a new folder.

7. Delete Folder:
• rmdir NewFolder: Deletes an empty folder.

8. Create File:
• type nul > newfile.txt: Creates an empty file.

9. Clear Screen:
• cls: Clears the terminal screen.

10. Exit Terminal:
• exit: Ends the command prompt session.

Hotkeys:

1. Clear Screen: Ctrl + L (may not work — in that case, use cls)

2. Command History: Use the up/down arrow keys to scroll through previous commands.

3. Interrupt Command: Ctrl + C

4. Auto-Complete: Tab

TERMINAL COMMANDS

If You're Using PowerShell on Windows:

Folders and Paths:

1. View Directory Contents:
• Get-ChildItem: Lists files and folders.

2. Change Directory:
• Set-Location C:\Users\Username\Documents: Navigates to the specified directory.

3. Copy Files:
• Copy-Item file.txt D:\Backup\file.txt: Copies the file to the specified location.

4. Delete Files:
• Remove-Item file.txt: Deletes the file.

5. Move Files:
• Move-Item file.txt D:\Backup\: Moves the file.

6. Create Folder:
• New-Item -ItemType Directory -Name NewFolder: Creates a new folder.

7. Delete Folder:
• Remove-Item NewFolder: Deletes the folder.

8. Create File:
• New-Item -ItemType File -Name newfile.txt: Creates an empty file.

9. Clear Screen:
• Clear-Host: Clears the terminal screen.

10. Exit PowerShell:
• Exit-PSSession: Ends the PowerShell session.

Hotkeys:

1. Clear Screen: Ctrl + L

2. Command History: Use the up/down arrow keys to browse previous commands.

3. Interrupt Command: Ctrl + C

4. Auto-Complete: Tab

Windows Terminal Supports all the commands listed above for both cmd.exe and PowerShell.

Permission Issue on macOS

— use sudo

Installation Error

— Sometimes on macOS, when installing packages or running terminal commands, instead of a successful result, you'll see an error.

Example:

Installing a new project via **npm create vite**

Often, this results in a permission error because your current user account doesn't have administrator privileges.

```
~/Desktop $npm create vite
```

```
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/a8/6a864be1a96b24ee1fba24b18579b130cbda44997f16ab53e7d076f0b9cd09d9d560fab2d141349fda59284c98ce721e61f4cda5ab0f7901b39cc112b59
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/sha512/8c/a8/6a864be1a96b24ee1fba24b18579b130cbda44997f16ab53e7d076f0b9cd09d9d560fab2d141349fda59284c98ce721e61f4cda5ab0f7901b39cc112b59'
npm error File exists: /Users/dmitrijfokeev/.npm/_cacache/content-v2/sha512/8c/a8/6a864be1a96b24ee1fba24b18579b130cbda444997f16ab53e7d076f0b9cd09d9d560fab2d141349fda59284c98ce721e61f4cda5ab0f7901b39cc112b59
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 react-dom react-scripts cra-template has failed.

Deleting generated file... package.json
Deleting pop1/ from /Users/dmitrijfokeev/Desktop
Done.
```

---

2 Ways to Solve It:

1. Use sudo Before the Command. This tells the system to run the command as an administrator. The terminal will prompt you for your password, then execute the command with elevated privileges. However: this may lead to issues later — for example, deleting or modifying files in the project might require additional admin permission prompts.

Example: `~/Desktop $sudo npm create vite`

2. Use the following command in the terminal to grant administrator-level permissions to your current user account (you can copy it from here or from the slide's comments):

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

JS Cheat Sheet

— A core reference guide for JavaScript

REACT JS

switch operator

— an alternative to if

— is a control flow construct in JavaScript that allows you to execute one of several possible operations depending on the value of an expression.

It’s especially useful when you need to compare a single value against multiple possible options and execute the corresponding code.

— You can copy the code from the comments below and test it out.

switch **operator:**

```
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
}
```

How switch Works?

1. Expression Evaluation:

• The expression inside the parentheses after switch is evaluated once.
• The result is then compared sequentially with the values listed in each case.

2. Code Execution:

• switch compares the expression’s value against each case value in order.
• Once a match is found, the corresponding block of code is executed.

3. Code Execution:

• When a match is found, the code inside the matching case block is executed.
• The break statement is used to stop the execution of subsequent case blocks. If break is omitted, switch continues executing the next case blocks, even after a match is found (this is called a “fall-through”).

4. The default Block:

• The default block is executed if none of the case values match the expression.
• This block is optional, but useful for handling all unspecified cases

ARRAY

Array methods:

Methods for adding and removing elements:

- **push():** Adds one or more elements to the end of an array and returns the new length of the array. array.push(element1, ..., elementN)
- **pop():** Removes the last element from an array and returns it. array.pop()
- **unshift():** Adds one or more elements to the beginning of an array and returns the new length of the array. (element1, ..., elementN)
- **shift():** Removes the first element from an array and returns it. array.shift()

Methods for iterating over elements:

- **forEach():** Executes a provided function once for each array element. array.forEach((item, index, arr) => {})
- **map():** Creates a new array with the results of calling a provided function on every element. array.map((item, index, arr) => {})
- **filter():** Creates a new array with all elements that pass the test implemented by the provided function. array.filter((item, index, arr) => {})
- **reduce():** Applies a function against an accumulator and each element (from left to right) to reduce it to a single value. array.reduce((accumulator, item, index, arr) => {}, initialValue)

Methods for searching elements:

- **find():** Returns the first element that satisfies the provided testing function. array.find((item, index, arr) => {})
- **findIndex():** Returns the index of the first element that satisfies the provided testing function. array.findIndex((item, index, arr) => {})
- **includes():** Determines whether an array includes a certain element, returning true or false. array.includes(element)
- **indexOf():** Returns the first index at which a given element can be found, or -1 if not found. array.indexOf(element)

Methods for modifying elements:

- **slice():** Returns a shallow copy of a portion of an array into a new array. array.slice(begin, end)
- **splice():** Changes the contents of an array by removing or replacing existing elements and/or adding new elements. array.splice(start, deleteCount, item1, ..., itemN)
- **concat():** Returns a new array that is the result of merging the array with other arrays and/or values. array.concat(value1, ..., valueN)
- **join():** Joins all elements of an array into a string. array.join(separator)

Methods for checking conditions:

- **every():** Tests whether all elements pass the test implemented by the provided function. array.every((item, index, arr) => {})
- **some():** Tests whether at least one element passes the test implemented by the provided function. array.some((item, index, arr) => {})

sort() method:

- Sorts the elements of an array alphabetically. The default sorting order corresponds to Unicode code points.

1-The sort() method modifies the original array.
Therefore, try to work with array copies.

For arrays containing strings, it sorts alphabetically (uppercase letters have higher priority).

2-The sort() method converts elements to strings, so digits will be sorted by string priority rather than numerically.

Therefore, to sort numbers you need to use a callback function with parameters a and b.

Using these parameters, all numbers in the array are compared.

- If number a is greater than number b, swap the elements.
- If number a is less than number b, keep the order as is.

ARRAY

About sort:

Sort method:

• Sorts the elements of an array in alphabetical order. The default sort order is based on Unicode code points.

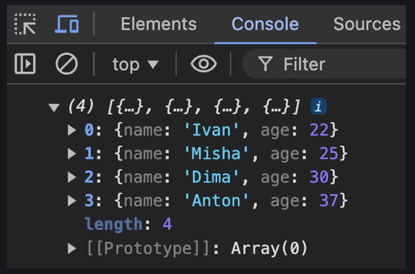
1-sort() Modifies the Original Array
ПThat’s why it’s recommended to work with a copy of the array.

For arrays containing only strings, it sorts alphabetically(Uppercase letters are prioritized over lowercase.).

```
const arr = ["d", "e", "b", "t", "r", "x", "s", "a", "g", "u"];

sortedArr = [...arr].sort();
console.log(sortedArr); // (10) ['a', 'b', 'd', 'e', 'g', 'r', 's', 't', 'u', 'x']
```

2- a and b are two elements of the array being compared during sorting. Each array element is an object containing the age property.
• The comparison function (a, b) => a.age - b.age determines the sorting order:
• If the result is less than zero, a should come before b.
• If the result is zero, the order of a and b remains unchanged.
• If the result is greater than zero, b should come before 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);
```



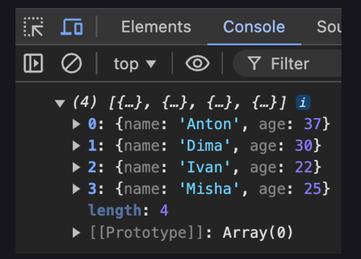
Create an independent copy of the array because the sort method modifies the original array, which is not recommended.

3. Alphabetical Sorting in an Object

To compare values properly, the localeCompare() method is needed.

localeCompare()
This is a method of the String object in JavaScript, used to compare two strings according to locale-specific sorting rules.

```
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);
```



The localeCompare() method compares the string it's called on with the provided string (method argument) and returns a numeric value:
• If the calling string should come before the provided string in sort order, it returns a negative number.
• If the strings are identical according to locale rules, it returns 0.
• If the calling string should come after the provided string, it returns a positive number.

2. The localeCompare() method takes into account locale and language-specific rules during comparison, including letter order, case sensitivity, and accents.

Create an independent copy of the array because the sort method modifies the original array, which is not recommended.

OBJECT

Object Methods:

Methods for Retrieving Object Information:

- **Object.keys() –** Returns an array containing the names of all enumerable properties of the object.
- **Object.values() –** Returns an array containing the values of all enumerable properties of the object.
- **Object.entries() –** Returns an array of key-value pair arrays for all enumerable properties of the object.

Methods for Working with Object Prototypes:

- **Object.create() –** Creates a new object with the specified prototype and properties.
- **Object.getPrototypeOf() –** Returns the prototype of the specified object.
- **Object.setPrototypeOf() –** Sets the prototype (i.e., the internal [[Prototype]] property) of the specified object.

Methods for Defining Object Properties:

- **Object.defineProperty() –** Defines or modifies a property directly on an object.
- **Object.defineProperties() –** Defines or modifies multiple properties on an object.
- **Object.getOwnPropertyDescriptor() –** Returns the property descriptor for a specific property.
  Object.getOwnPropertyDescriptors() – Returns all property descriptors for an object.

Methods for Working with Object Properties:

- **Object.hasOwn() –** Checks if a property exists directly on the object (not inherited).
- **Object.propertyIsEnumerable() –** Returns true if the specified property is enumerable using for...in.
- **Object.is() –** Determines whether two values are the same.

Methods for Creating and Cloning Objects:

- **O bject.assign() –** Copies enumerable properties from one or more source objects to a target object.
- **Object.freeze() –** Freezes the object so properties can’t be changed or deleted.
- **Object.seal() –** Seals the object, preventing new properties and making existing ones non-configurable.

Methods for Working with Object Metadata:

- **Object.isExtensible() –** Checks if the object is extensible (i.e., if new properties can be added).

- **Object.isFrozen() –** Checks if the object is frozen.

- **Object.isSealed() –** Checks if the object is sealed.

- **Object.preventExtensions() –** Prevents new properties from being added to the object.

STRING

String Methods:

Methods for Searching and Extracting Substrings:

- **charAt() –** Returns the character at a specified index.
- **charCodeAt() –** Returns the Unicode value of the character at a specified index.
- **concat() –** Combines the specified strings into one.
- **includes() –** Checks if one string is found within another.
- **indexOf() –** Returns the index of the first occurrence of a specified value or -1 if not found.
- **lastIndexOf() –** Returns the index of the last occurrence of a specified value or -1 if not found.
- **localeCompare() –** Compares two strings according to the current locale and returns a number indicating their order.
- **match() –** Retrieves the result of matching a string against a regular expression.
- **matchAll() –** Returns an iterator of all results matching a regular expression, including capture groups.
- **search() –** Executes a search for a match between a regular expression and the string.
- **slice() –** Extracts a section of the string and returns it as a new string.
- **split() –** Splits a string into an array of substrings using a specified separator.
- **substring() –** Returns a substring between two indexes or from a start index to the end.
- **substr() –** Returns a substring starting at a specified index with a specified length.

Methods for Modifying Strings:

- **toLowerCase() –** Converts the string to lowercase.
- **toUpperCase() –** Converts the string to uppercase.
- **trim() –** Removes whitespace from both ends of the string.
- **trimStart() / trimLeft() –** Removes whitespace from the beginning of the string.
- **trimEnd() / trimRight() –** Removes whitespace from the end of the string.
- **padStart() –** Pads the beginning of the string to the specified length.
- **padEnd() –** Pads the end of the string to the specified length.
- **repeat() –** Returns a new string with a specified number of copies.
- **replace() –** Replaces a substring or pattern match with a new value.
- **replaceAll() –** Replaces all instances of a substring or pattern match with a new value.

Methods for Unicode and Code Points:

- **codePointAt() –** Returns the Unicode code point at a given position.
- **normalize() –** Returns the normalized form of the string according to Unicode specification.

Pattern-Related Methods:

- **startsWith() –** Checks if a string begins with specified characters.
- **endsWith() –** Checks if a string ends with specified characters.

Number Methods:

Methods of the Number Object:

- **Number.isFinite() –** Determines if the passed value is a finite number.
- **Number.isInteger() –** Determines if the passed value is an integer.
- **Number.isNaN() –** Determines if the passed value is NaN (Not-a-Number).
- **Number.isSafeInteger() –** Checks if the passed value is a safe integer (within the safe integer range).
- **Number.parseFloat() –** Converts a string to a floating-point number.
- **Number.parseInt() –** Converts a string to an integer using the specified radix.

Number prototype Methods:

- **toExponential() –** Returns a string representing the number in exponential notation.
- **toFixed() –** Returns a string with the number rounded to a fixed number of decimal places.
- **toLocaleString() –** Returns a string with the number formatted according to locale.
- **toPrecision() –** Returns a string with a number formatted to a specified precision.
- **toString() –** Returns a string representing the number in the specified base.
- **valueOf() –** Returns the primitive value of a Number object.

Global Number Methods:

- **isNaN() –** Determines if the value is NaN.
- **isFinite() –** Determines if the value is a finite number.
- **parseInt() –** Parses a string and returns an integer using the specified radix.
- **parseFloat() –** Parses a string and returns a floating-point number.

Properties of the Number Object:

- **Number.EPSILON –** The smallest interval between two representable numbers.
- **Number.MAX_SAFE_INTEGER –** The maximum safe integer in JavaScript.
- **Number.MAX_VALUE –** The largest representable number in JavaScript.
- **Number.MIN_SAFE_INTEGER –** The minimum safe integer in JavaScript.
- **Number.MIN_VALUE –** The smallest positive number in JavaScript.
- **Number.NaN –** Represents the Not-a-Number value.
- **Number.NEGATIVE_INFINITY –** Represents negative infinity.
- **Number.POSITIVE_INFINITY –** Represents positive infinity.

DATE

Date Methods:

Creating and Getting the Current Date:

- **new Date() –** Creates a Date object representing the current date and time.
- **Date.now() –** Returns the number of milliseconds since January 1, 1970 (Unix epoch).

Creating a Date Object with a Specific Date and Time:

- **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)

Getting Date Components:

- **getFullYear() –** Returns the four-digit year.
- **getMonth() –** Returns the month (0–11, where 0 = January).
- **getDate() –** Returns the day of the month (1–31).
- **getDay() –** Returns the day of the week (0–6, where 0 = Sunday).
- **getHours() –** Returns the hour (0–23).
- **getMinutes() –** Returns the minutes (0–59).
- **getSeconds() –** Returns the seconds (0–59).
- **getMilliseconds() –** Returns the milliseconds (0–999).
- **getTime() –** Returns the time value in milliseconds since January 1, 1970.
- **getTimezoneOffset() –** Returns the difference in minutes between local time and UTC.

Setting Date Components:

- **setFullYear(year, [month], [day]) –** Sets the year; optionally the month and day.
- **setMonth(month, [day]) –** Sets the month; optionally the day.
- **setDate(day) –** Sets the day of the month.
- **setHours(hours, [minutes], [seconds], [milliseconds]) –** Sets the time.
- **setMinutes(minutes, [seconds], [milliseconds]) –** Sets the minutes, seconds, and milliseconds.
- **setSeconds(seconds, [milliseconds]) –** Sets the seconds and milliseconds.
- **setMilliseconds(milliseconds) –** Sets the milliseconds.
- **setTime(milliseconds) –** Sets the time in milliseconds since January 1, 1970.

Converting Dates to Strings:

- **toDateString() –** Returns a string containing only the date.
- **toTimeString() –** Returns a string containing only the time.
- **toLocaleDateString() –** Returns the date in localized format.
- **toLocaleTimeString() –** Returns the time in localized format.
- **toLocaleString() –** Returns the date and time in localized format.
- **toISOString() –** Returns a string in ISO 8601 format.
- **toUTCString() –** Returns a string representing the date in UTC format.
- **toJSON() – R**eturns a string representing the date in JSON format (same as toISOString).

Destructuring:

Array Destructuring

Basic Syntax:

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

Skipping Elements:

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

Rest Elements:

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

Default Values:

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

Object Destructuring

Basic Syntax:

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

Variable Renaming:

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

Rest Elements:

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

Default Values:

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

Destructuring in Functions

Function Parameters:

```
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
```

Default Values in Parameters:

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

Examples of Complex Destructuring:

Nested Structures:

```
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
```

Combined Examples:

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

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

Nested Destructuring:

Arrays in Objects:

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

Objects in Arrays:

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

Asynchronicity

Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation. It can be either resolved (fulfilled) or rejected (failed).

```
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(): Called when a promise is successfully resolved. It takes two arguments: functions that are called upon success and error respectively.

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

catch(): Called when a promise is rejected. It is a shorthand for then(null, onRejected).

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

finally(): Called in all cases when the promise settles (regardless of whether it was fulfilled or rejected).

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

Promise Chaining

```
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");
  });
```

Asynchronicity

Async/ Await

async и await — async and await are syntactic sugar over Promises that make asynchronous code more readable and linear.

async: Declares an asynchronous function that returns a Promise.
await: Waits for a Promise to resolve and returns its result. Can only be used inside an async function.

Error Handling

Errors in async functions are handled using **try...catch.**
finally - will run in any case, regardless of whether the Promise was fulfilled or rejected.

Promise.all()

Use P**romise.all()** to run multiple promises in parallel. Use finally to run code after all of them finish.

```
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();
```

```
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();
```

Closures

— When a function returns another function.

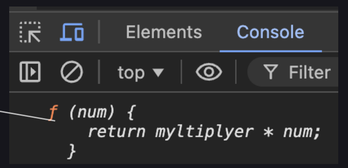
Example of Calling a Function That Returns Another Function

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

The console.log is just for demonstration purposes.

The number 2 is just an example for now; it doesn't affect anything yet.

We get the inner function in the console — makes sense.



You can do it like this.

THIS IS WHAT A FUNCTION CLOSURE IS.

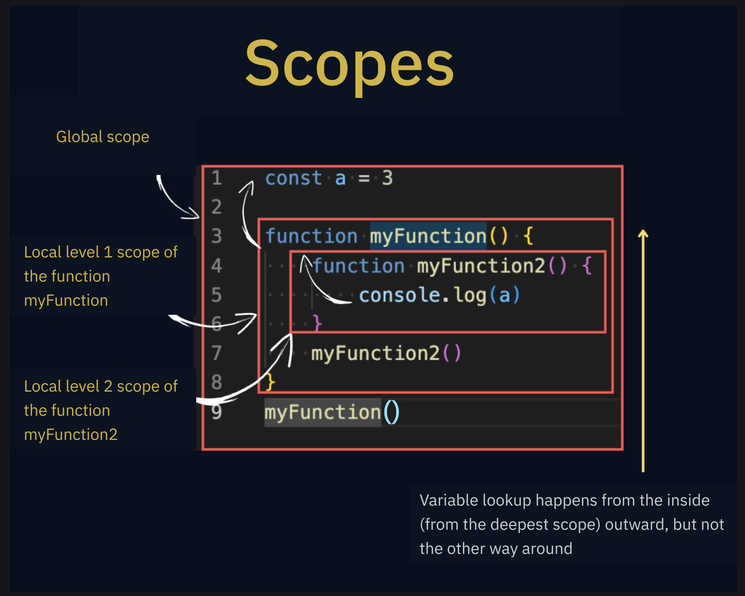
```
38  function multiplyFn(myltiplyer) {
39    return function (num) {
40      return myltiplyer * num;
41    };
42  }
43
44  console.log(multiplyFn(4)(2)); /// 8
```

Calling multiplyFn(4)(2) will first invoke the outer function with the argument 4.

It then returns the inner function and immediately calls it with the argument 2.

Result: 8
Because 4 * 2 = 8

Scope Reference Guide:



A closure occurs when an inner function remembers and has access to variables from its lexical scope, even after the outer function that contains it has finished executing.

When an inner function is created inside an outer function and uses variables declared in that outer function, a closure is formed — the inner function retains access to the outer function's variables.

YOU CAN CLOSE OVER VALUES LIKE THIS

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

BUT IT'S MORE CONVENIENT THIS WAY:

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

We create a function expression (assign the function to a variable):

We call the function expression with the argument 7.

We closed over the function multiplyFn(2) and its argument 2.

(Now, every time multiplyByTwo(number) is called, the number passed to multiplyByTwo will be multiplied by 2.)

Here, we've closed over (preserved) the multiplyer parameter of the multiplyFn

When an inner function is created inside an outer function and uses the variables and parameters declared in the outer function, a closure is formed.

CLOSURE STEP BY STEP:

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

1

We created a function multiplyFn
with the parameter (multiplyer).

2

The function multiplyFn does one thing — it
creates and returns a new function (an
anonymous one, meaning it has no name).

3

A function expression multiplyByTwo is then created.

3The value of the variable multiplyByTwo is the result of executing multiplyFn with the argument 2.

The result of calling multiplyFn(2) is its inner function.

Therefore, the function expression multiplyByTwo is actually the anonymous inner function of multiplyFn, with the closed-over value of multiplyFn's parameter — which in this case is the number 2.

4

Finally, we call the function expression multiplyByTwo with the parameter 7.

That means the inner anonymous function is now executed with the argument 7.
All this function does is return the result of multiplying its own argument (num) by the outer function's argument (multiplyer).
The result is 14.

ANOTHER CLOSURE EXAMPLE — STEP BY STEP

```
function outer() {
    let count = 0; // Local variable of the outer function
    function inner() {
        count++;
        console.log(count);
    }
    return inner;
}

const myFunction = outer(); // myFunction becomes a closure
myFunction(); // Outputs 1
myFunction(); // Outputs 2
```

-outer declares the variable count and the function inner.

- inner increments count and logs it. Even though outer has finished executing after the call myFunction = outer(), inner still "sees" count.

- inner retains a reference to outer’s lexical environment, allowing it to access and modify count.

Form Reference **Guide**

— Creating HTML Forms

Basic Form Structure

- action: The URL the form will be submitted to.
- method: The HTTP method used to send the form (GET or POST).

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

Text Fields & Input Elements

- text: Standard single-line input.
- password: Input for passwords, characters are hidden.
- email: Email input with built-in validation..
- textarea: Multiline text input.

```
<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>
```

Checkboxes and Radio Buttons

- checkbox: A toggle that can be checked or not.
- radio: A selection where only one option can be chosen among a group.

```
<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
```

Dropdown Lists and Groups

- select and option: Used to create dropdown menus.

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

Buttons

- submit: Submits the form.
- button: A general-purpose button, usually used with JavaScript.
- reset: Resets the form fields.

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

Hidden Fields

- hidden: Not visible to the user, but holds important form data.

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

Validation Attributes

- required: The field must be filled before submitting.
- pattern: A regular expression the input must match.
- min и max: Set numeric range limits.
- maxlength и minlength: Set text length constraints..

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

Good Practices

- Use <label> to enhance accessibility:

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

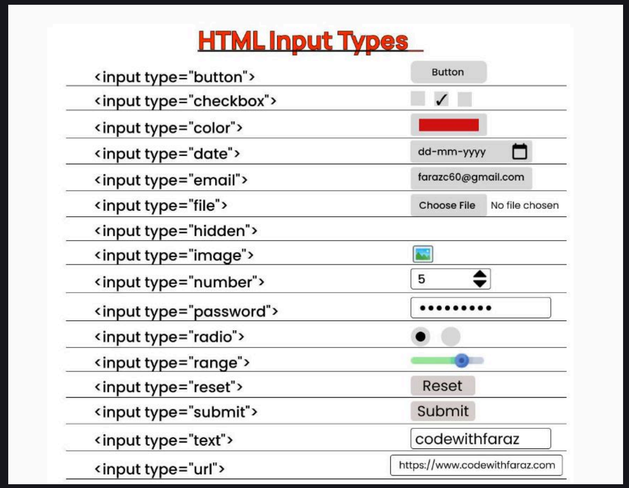
- Group related inputs with <fieldset> and <legend>:

```
<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>
```

Example of a Complex Form

```
<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>
```

Input Type Reference



Creating **a New** **React** Project

— vite

Vite

— It is a modern build tool for JavaScript projects.
Its main goal is to provide fast and efficient development, especially for modern frameworks such as Vue, React, Svelte, and others.

How to Create a New Vite Project:

1. Open a Directory

Open your terminal and navigate to the folder where you want to create the project (use the terminal in VS Code or your system terminal).

2. Run the Command:

npm create vite

Set a name for your project, choose a framework , choose the language.

3. Install Packages:

Open the project folder in VS Code and run:

npm install

4. Configure ESLint (Optional)

If needed, set up ESLint and adjust other project configurations.

5. Clean Up (Optional)

Remove any unnecessary files, folders, or demo code.

6. Start the Project

In the terminal, run:

npm run dev

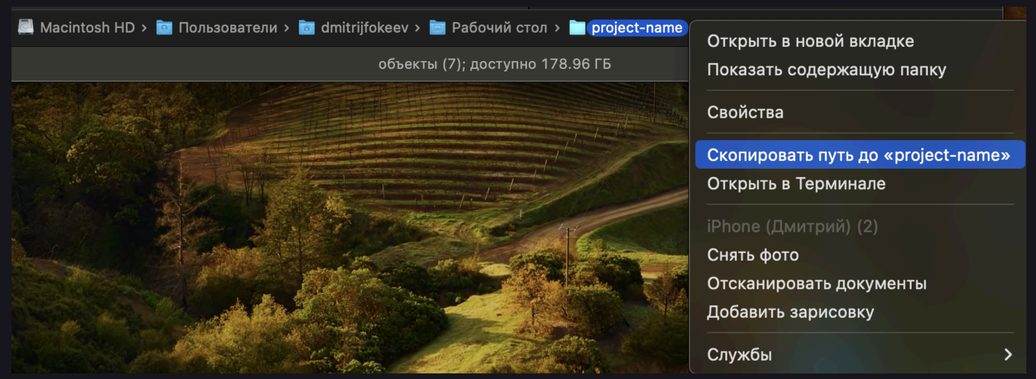
Copy the link that appears and open it in your browser.

7. Stop the Project

In the terminal, press Ctrl + C.

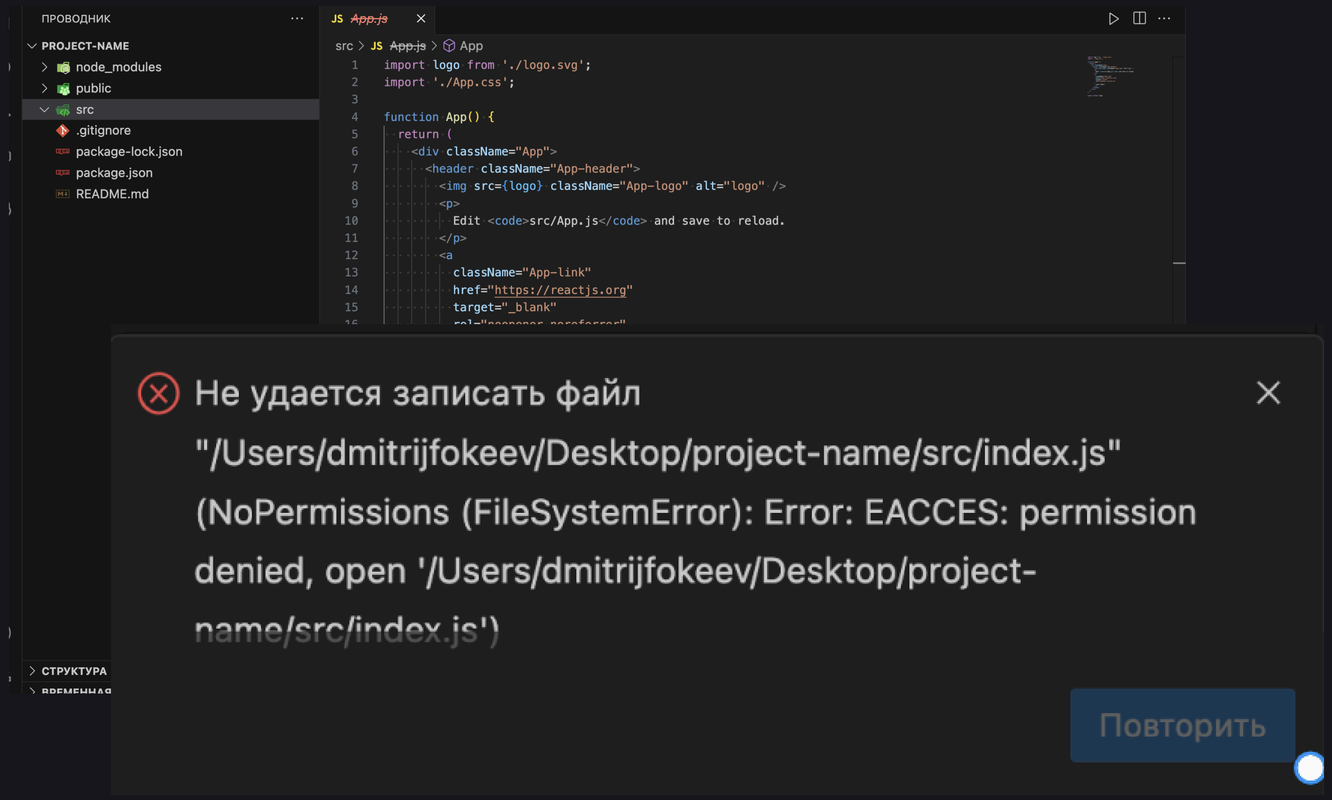
Issue with Deleting or Creating Files in a Project on macOS:

Copy the path to the project folder

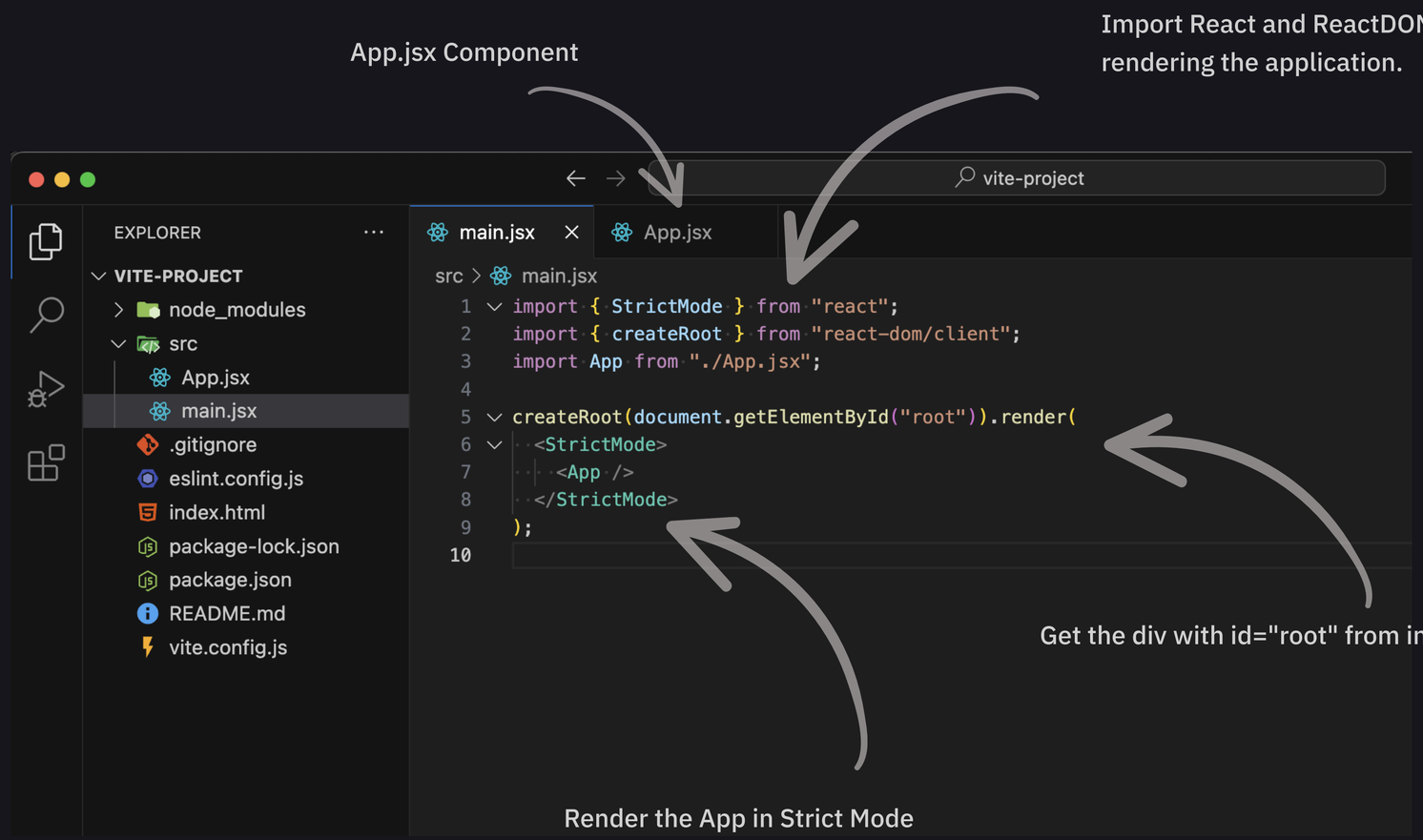


Then enter the following command in the terminal:

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



Basic Structure of a React Application:



React.createElement

— function React

React.createElement

— It’s the function that React uses to create elements.

React.createElement

This function is the foundation of the entire React element concept, and it's what React uses to build and update the virtual DOM.

Without JSX, you could write a component like this:

```
function MyComponent() {
  return React.createElement(
    'div',                  // Element type (HTML tag 'div')
    { className: 'my-div' },    // Props (here class 'my-div')
    'Hello, world!'    //  Child element (string 'Hello, world!')
  );
}
```

This is equivalent to the following JSX:

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

When you write JSX like Hello, world! , it compiles down to a call to React.createElement.
JSX is syntactic sugar that makes writing React elements easier and more readable.
But under the hood, React always uses React.createElement to construct the element object.

React.createElement

— More on How It Works

Syntax of React.createElement

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

Arguments:

1. type:

- The type of element to create.
- A string representing an HTML tag (e.g., 'div', 'span').
- A function or class for a React component.

2. props:

- An object containing all the element's attributes (props). If there are no props, pass null.
- Example: { className: 'my-class', id: 'my-id' }.

3. children:

- The children of the element. Can be:
- Text.
- Other React elements.
- An array of elements.

Example with Props and Children:

```
const element = React.createElement(
  'div',    // Element type
  { className: 'my-class', id: 'my-id' },    // Props
  'Hello, ',  //  // First child element (text)
  React.createElement('strong', null, 'world!')    // <strong>World!</strong> → // Second child element — <strong>World!</strong>
);
```

Same Code Using JSX:

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

JSX

— JavaScript XML

JSX (JavaScript XML) — It is a language extension. JavaScript

— Which allows you to write elements that look like HTML or XML directly in your React code.

Simply put, JSX is a syntax that lets you write HTML, CSS, and JS all in one place.

JSX Syntax Without JavaScript

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

JSX Syntax With JavaScript

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

Example of a React Component with JSX:

```
function UserProfile({ user }) {
  return (
    <div className="user-profile">
      {/* Using curly braces to embed JS expressions */}
      <h1>Welcome, {user.name}!</h1>
      <p>Age: {user.age}</p>
      {/* Conditional rendering in JSX */}
      {user.age >= 18 ? (
        <p>You are an adult.</p>
      ) : (
        <p>You are under 18.</p>
      )}
    </div>
  );
}

// Example of using the UserProfile component
const user = {
  name: "Alice",
  age: 22
};

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

JSX Under the Hood

— Or, how it would look if we used methods instead of JSX

“Under the hood,” JSX is transformed into calls to React.createElement().
For example, this JSX code:

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

Is transformed into:

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

The React.createElement() function takes three arguments:

1. Element Type (e.g., 'h1', 'div', or a React component).
2. Props Object, Attributes of the element, such as className, id, style. Can be null
3. Children of an element can be text, other elements, or JavaScript expressions.

JSX - **CSS**

— Work with CSS

CSS in JSX

— How CSS Works in a React Component

Styles are written as a JavaScript object, where the keys are camelCase versions of CSS properties, and the values are either strings or numbers. For numeric values like width, padding, etc., the default unit is pixels.

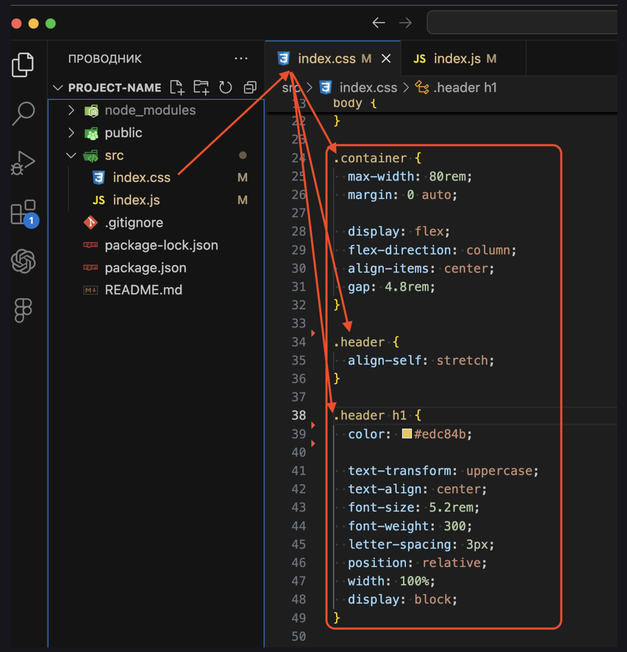
Values such as colors, URLs, and other CSS properties that are typically written in quotes must also be specified as strings.

You can create a separate variable whose value is a style object, and then apply it to an element using the style attribute.

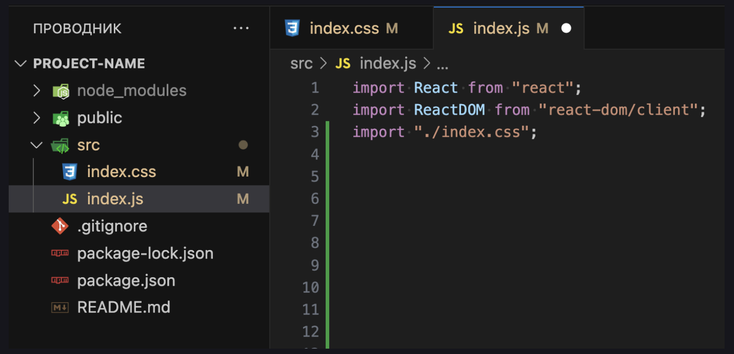
```
60  function Header() {
61    const style = {
62      color: "red",
63      backgroundColor: "black",
64      padding: 20,
65      margin: "10px 5px 15px 20px",
66      border: "1px solid blue",
67    };
68
69    return <h1 style={style}>Header H1</h1>;
70  }
```

index.css

— Using a CSS file with predefined styles for specific classes



Connecting index.css to index.js



CSS classes are used in JSX via the className attribute, because class is a reserved word in JavaScript.

---

index.css file

```css
.header {
  color: #edc84b;
  text-transform: uppercase;
  text-align: center;
  font-size: 5.2rem;
  font-weight: 300;
  letter-spacing: 3px;
  position: relative;
  width: 100%;
  display: block;
}
```

index.js file

```jsx
function Header() {
  return <h1 className="header">Header H1</h1>;
}
```

JSX - **Quick Reference**

— How It Differs from HTML / CSS / JS

General Rules of JSX:

• Component names in React must start with an uppercase letter to distinguish them from HTML tags.

• JSX works almost like HTML, but you can insert “JavaScript mode” using {} — to reference variables, create arrays or objects, use .map(), the ternary operator, or any other valid JS expressions.

• Statements like if, else, for, switch are not allowed inside {} in JSX.

• For conditional rendering, use ternary operator or logical AND (&&) instead of if...else.

• Comments in JSX: Use this format inside JSX blocks:
{/* This is a comment */}

Differences Between JSX and HTML

• Components can be used inside {} since they are JavaScript expressions.

• Inline CSS styles are written as style={object}.

• CSS property names are written in camelCase (e.g., backgroundColor, fontSize).

• Use className instead of class.

• Every tag must be closed, even self-closing ones like <img /> or
.

• Event handlers and props must use camelCase, such as onClick, onMouseOver.

• aria- and data- attributes are written with dashes, just like in regular HTML.

Paths in React Projects

— vite

Relative Path

— Relative paths point to a file's location relative to the current folder, using:
./ (current directory)
or
../ (one level up)

Absolute Path

— This is a full path that points to the exact location of a file or resource starting from the root.
An absolute path is always complete and does not depend on the current file’s location.

React Specifics

1. Absolute Path with Aliases:

If the folder structure changes, absolute imports remain stable.
You can use aliases in React (e.g., @) to point to the src root.

2. public Folder:

Files in the public directory are accessible with absolute paths — no need to write relative paths.
For example: /img/some.jpg instead of ./public/img/some.jpg

3. Exporting Modules from src

All files and components in the src folder should be exported using export or export default, so they can be imported elsewhere.

Modules:

— Every file in JavaScript is treated as a module.

Modules can export their functions, components, or variables to make them accessible to other files.

Export:

This is the process that allows you to “send out” a part of your code (a function, variable, or component) from the current file so other files can import and use it.

Types of Export:

Default Export (export default):

- Allows you to export one main item per file.
- When importing, you don’t use curly braces, and you can name it however you like.
- Commonly used for exporting the primary component of a file.

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

Named Export (export):

- Allows you to export multiple items from one file.
- When importing, you must use curly braces, and the names must match.
- Often used for utility functions, helpers, or when exporting multiple values.

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

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

Import:

This is the process of bringing in code that has been exported from another file so it can be used in the current file.

```
// App.js
import Button from './Button';
```

```
// App.js
import { add, subtract } from './utils';
```

Props

— props (abbreviation of properties)

Props

— A way to pass data from parent components to child components.

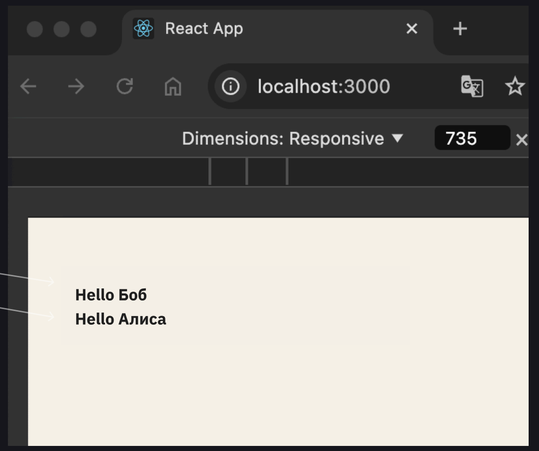
Props make components more reusable and modular. Components receive props as a function parameter and use them to display data or adjust their behavior depending on the input.

Parent component using the <Welcome /> component as a child.

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

Child component.

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



In this case, the App component uses the Welcome component twice, passing different names as props. Each instance of the Welcome component will display a greeting with the corresponding name.

Props

— A way to pass data from parent components to child components.

Important rules:

The parent component, in which the child component is used, is the control center that sends instructions on what the element should be, with which properties, and in what quantity.

The child component is the description of the element (a template): what its structure is and what properties it can have.

In React, data is passed from the parent component to the child through props. The opposite is not possible.

The child component receives data as immutable properties (props), which it can use but cannot modify.

map()

— map() in components

map()

— Of course, manually writing out each property is not the best option.

That’s why, as a rule, we have an array of objects with the necessary data.

We can pass this data using props.

```
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()

To use data from objects in our components, we need to iterate over them with the standard map() method.

```
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. The parent component iterates through an array of objects using map().
2. Each object becomes a separate instance of the component.
3. Each component instance is given a property, and data from the object (such as product.id, product.name, etc.) is passed into it.
4. A child component is created that accepts properties (props) as function parameters. This component defines the structure and the places where those properties are used.

Thus, from the “control center” (the parent component), instructions are given to create as many copies of the template (child component) as there are elements in the array, and to use the data from the array elements in the specified places of the template (child component).

```
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()

Let’s slightly simplify the iteration in the parent component by passing the entire object to the child, rather than its individual properties.

(Compare the changes with the previous slide and experiment in the code.)

```
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>
  );
}
```

```
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>
  );
}
```

Destructuring props

— Let’s make it even simpler and more readable :)

Destructuring props

Instead of extracting the object from the props passed into the component, we can immediately destructure the props in the component's parameters.

Version without destructuring

```jsx
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>
  );
}
```

Version with destructuring

```jsx
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>
  );
}
```

Do not assign the object in the component’s parameters.

The reason why React components usually don’t use the = sign in props during destructuring is that React automatically handles props and passes them into the component.

This is not the correct way to destructure.

```
function ProductCard({ objectInProps } = props) {
  // Function body
}
```

Correct way

```
function ProductCard({ objectInProps }) {
  // Function body
}
```

Alternatively, you can do it this way too

```
function ProductCard(props) {
  const { objectInProps } = props;
  // Function body
}
```

map()

Let’s simplify the iteration in the parent component a bit by passing the entire object to the child, rather than its individual properties.

(Compare the changes with the previous slide and experiment in the code.)

```
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>
  );
}
```

```
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>
  );
}
```

The && operator

— rendering based on state

The && operator

false && true → returns the first false

true && true → returns the second true

The && operator in React is often used for conditional rendering of components or elements.

This allows you to display a component or element only when a certain condition is true.
The && operator returns the first operand (left) if it is falsy (false, null, undefined, 0, NaN, or an empty string ""), and the second operand (right) if the first is truthy (true).

```
// Component that displays the opening message
function OpenMessage() {
  return <h1>We are open! Come visit us.</h1>;
}

// Main application component
function App() {
  const now = new Date(); // Get the current time
  const hour = now.getHours(); // Extract the hour from the current date

  // Set display conditions
  const isOpen = hour >= 9 && hour < 17; // Open from 9 to 17 hours

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

If it’s currently 18:00, then isOpen is false.
If isOpen = false → render “We are Closed”

If isOpen = true → render “We are Open” (the second operand)

If both before and after && are true, the second value is returned and rendered — for example, the <OpenMessage /> component.

The ternary operator

— condition ? expressionIfTrue : expressionIfFalse

The ternary operator

condition ? expressionIfTrue :
expressionIfFalse

The ternary operator is a shorthand form of the if...else conditional.

If the condition is true → do this ? : if not → do that.

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

Ternary operator in React

condition ? expressionIfTrue : expressionIfFalse

In JSX you cannot use if statements, because JSX is not JavaScript itself but syntactic sugar for function calls that must return values.

However, you can use the ternary operator inside JSX:

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

Using if in React

Although if cannot be used directly inside JSX, you can use it in the component function before returning 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>
  );
}
```

Conditional Rendering

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

REACT JS

Conditional Rendering

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

Data Objects

```
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,
  },
];
```

Components

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

function ProductCard({ product }) {
  // Conditional rendering based on product availability
  if (!product.isAvailable) {
    return null;  // Do not render the card if the product is unavailable
  }

  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: This component receives a product object as a prop.

At the start of the component, there is a condition that checks the isAvailable property.
If the product is unavailable (isAvailable equals false), the function returns null, which prevents rendering of that product card.

ProductList: A component that displays a list of all products.
It uses ProductCard to render each product, passing the product data into it.

<React.Fragment>
</React.Fragment>

or

<>
</>

<>

</>

React Fragment allows you to group multiple JSX elements without creating extra DOM nodes — for example, without wrapping everything in an additional .

Instead of this

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

This

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

There will be no extra in the markup.

Or like this
It’s the same thing

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

Event Handling

— onClick = {()=> alert('Click')}

onClick = {function}

In React, event handling such as button clicks is implemented using attributes that start with on, like onClick.

A separate function that will be triggered by an event (for example, a click).

```
function handleClick() {
  alert( 'The button was clicked!'  );
}
```

Assign this function to the onClick attribute of a button component. In React, this is done inside JSX code:

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

You can also use arrow functions to automatically bind context or define the handler directly inside JSX:

```
<button onClick={() => alert( 'The button was clicked!'  )}>
  Нажми на меня
</button>
```

Main Events

List of main events

Mouse Events
- onClick: Left mouse button click
- onContextMenu: Right mouse button click to open context menu
- onDoubleClick: Double mouse click
- onDrag / onDragEnd / onDragEnter / onDragExit / onDragLeave / onDragOver / onDragStart / onDrop: Drag-and-drop events
- onMouseDown / onMouseEnter / onMouseLeave / onMouseMove / onMouseOut / onMouseOver / onMouseUp: Mouse interaction events

Keyboard Events
- onKeyDown / onKeyPress / onKeyUp: Key press events

Form Events
- onChange: Value change in a form element (e.g., typing or selecting a value)
- onInput: User input in a field
- onSubmit: Form submission

Focus Events
- onFocus: Element gains focus
- onBlur: Element loses focus

Touch Events
- onTouchCancel / onTouchEnd / onTouchMove / onTouchStart: Touch interactions on mobile devices

UI Events
- onScroll: Scrolling an element or page

Mouse Wheel Events
- onWheel: Mouse wheel scrolling

Image Events
- onLoad: Image finished loading
- onError: Error loading an image or file

Other Events
- onSelect: User selects text
- onAnimationStart / onAnimationEnd / onAnimationIteration: CSS animation events
- onTransitionEnd: End of a CSS transition

useState

— State Management

useState

This is one of the React hooks that allows components to have their own state

— You can copy the code from the comments below and test it.

The useState hook takes an initial state and returns an array with two elements:

1. The current state value.
2. A function to update that state.

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

Пример:

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

function Counter() {
  // Initializing the counter state with value 0
  const [count, setCount] = useState(0);

  // Function to increment the counter
  function increment() {
    setCount(count + 1);
  }

  return (
    <div>
      <p> You clicked {count} times </p>
      <button onClick={increment}>
        Click me
      </button>
    </div>
  );
}
```

Destructuring

The setCount function increases the count state by +1.

The state update function is triggered by clicking the button.

useState(0) initializes count with the value 0.

• setCount is the function that updates the value of count. Calling this function automatically re-renders the component with the new state value.

• A button in the component calls the increment function on click, which increases the value of count by one.

State Update

State can only be changed through setState().

In React, you cannot update state directly, because React uses a special mechanism to track state changes, which is based on the setState method.

1. Automatic rendering management

React tracks when state changes through setState (or setCount with useState) and knows it needs to re-render the component. If you update state directly, React won't detect the change and won't re-render, which means the UI won't display the updated data.

2. Immutability of state

React relies on immutable state for better performance. When state is updated via setState, React creates a new copy of the state instead of modifying the existing one. This allows React to quickly detect changes by comparing the new and old state.

3. Asynchronous updates

State updates in React can be asynchronous, and they are often batched for performance. setState ensures React processes updates in order and with the latest data.

If you modify state directly, you risk incorrect data because React won't be aware of the changes and won't be able to properly manage the update queue.

useState **call-back**

— Use a callback when updating the current value.

useState call-back

Use a callback to update state when the new state depends on the previous one

— You can copy the code from the comments below and test it.

BEFORE

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

function Counter() {
  // Initializing the counter state with value 0
  const [count, setCount] = useState(0);

  // Function to increment the counter
  function increment() {
    setCount(count + 1);
  }

  return (
    <div>
      <p> You clicked {count} times </p>
      <button onClick={increment}>
        Click me
      </button>
    </div>
  );
}
```

AFTER

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

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

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

  return (
    <div>
      <p> You clicked {count} times </p>
      <button onClick={increment}>
        Click me
      </button>
    </div>
  );
}
```

When you use the callback form setCount((count) => count + 1), you pass a function that receives the current state value as an argument and returns the new value.

This ensures you’re working with the most up-to-date state at the moment of the update, which is especially important in cases where state might change multiple times in a short period.

When to use **UseState**

— and how exactly

UseState

When to use

What useState does

1. Preserves values between renders: A value created with useState is retained across re-renders of the component.

2. Triggers re-rendering on change: When the state is updated through setState, the component automatically re-renders to display the new data.

3. Maintains immutability: Updating state with setState creates a new version of the state, allowing React to compare old and new values for optimized rendering.

Use useState when you need to

1. Track state for rendering: If a value created with useState changes, the component automatically re-renders, showing the updated value in the UI.

2. Preserve values between renders: Variables declared with useState keep their value across renders, unlike normal variables which reset.

THE DATA OF A REGULAR VARIABLE DOES NOT CHANGE DURING THE PROGRAM'S EXECUTION....

No

REGULAR VARIABLE

Yes,
they do change

WE CREATE A USESTATE.

WHEN NEW DATA OF A USESTATE VARIABLE DEPENDS ON ITS OLD DATA

Mo

```
const [newNumber, setNumber] = useState(1);
<button onClick={setNewNumber(10)}></button>
```

Yes

```
const [newNumber, setNumber] = useState(1);
<button onClick={setNewNumber((newNumber) => newNumber + 10)}></button>
```

The new value is obtained using the previous one

React **is Declarative**

— But what does that mean?

Declarative and Imperative

- Declarative code describes what should happen and focuses more on the final result.

- Imperative code describes how to make it happen, focusing on the step-by-step process.

Declarative and Imperative

— these are two approaches to writing code that describe different styles of interacting with programs. These styles are often mentioned in programming and UI development, especially in React and other modern libraries.

1. Declarative

The declarative style focuses on what the program should do, not how to do it. In declarative code, we describe the desired end result, and the system decides how to achieve it. This is a higher level of abstraction, making code easier to write and understand since it’s closer to natural language.

In React, using JSX, we write declarative code to describe how the UI should look:

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

2. Imperative

The imperative style focuses on how the program should perform a task. The developer provides exact instructions, describing the process step by step. This is a lower level of abstraction, where the developer manages specific operations.

Example in plain JavaScript

If we want to display the text “Hello, World!” on the screen in plain JavaScript, it might look like this:

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

Array.from()

— Creating an array using Array.from()

Automatically

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

It allows you to create a new array from an iterable or an array-like object.

Manually

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

The Array.from() method has several parameters:

1. First parameter – an array-like object. This object doesn't contain actual values, but it has a length property set to 20, which lets Array.from know how many elements should be in the resulting array.

2. Second parameter – a mapping function, called for each index of the new array. This function takes two arguments:

- _ (or any other name you prefer) – the value of the element (not used here because the source doesn't have real elements).
- i – the index of the current element.

How it works in your code:

The mapping function (_, i) => i + 1 generates values for each element of the new array.
i starts at 0 and goes up to 19 (since array indices in JavaScript start at 0). By adding 1 to each index, you get numbers from 1 to 20. Result: [1, 2, 3, ..., 20]

props.**children**

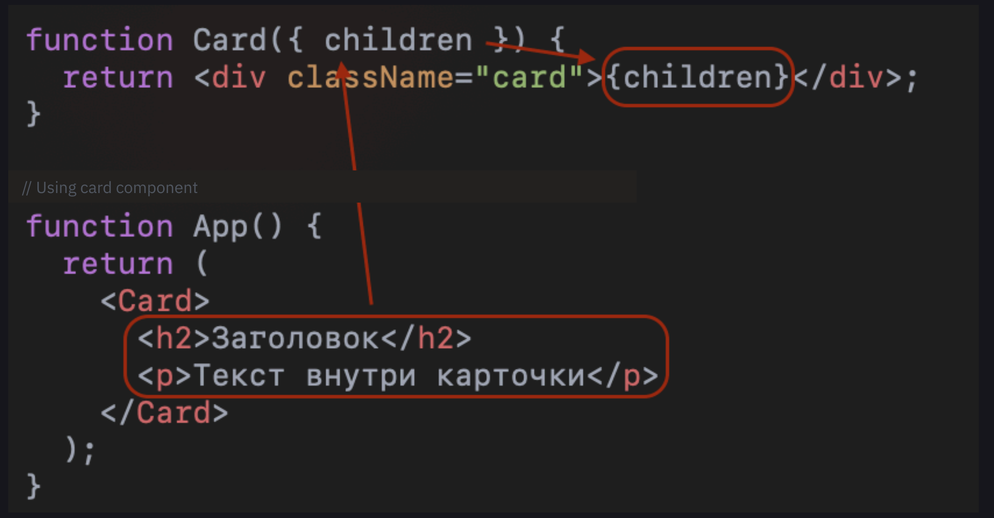
— In React, children is a special prop that allows passing child elements into a component.

props.children

children is a regular prop that is passed into a component. Its content is everything that is inside the opening and closing tags of the component.

In this example, <h2> and <p> are passed as the children prop into the Card component. We can use this prop in JSX inside the component itself.

Thus, everything we place between the opening and closing tags of the component when using it will be passed into the children prop.



THE KEYWORD CHILDREN IS RESERVED!

children can be:

As a primitive: a string, a number, a boolean value (for example, "Hello", 123, true).

- **A React element:** one element or multiple elements.
- **A function:** for example, for implementing render props.
- **An array:** a list of elements.

You can set a default value for children:

```
function DefaultContainer({ children = <p>        Default content        </p> }) {
  return <div>{children}</div>;
}
```

props.**without value**

— Always true

props.without value

In React, if you pass a prop (for example, isDisabled) without a value, it is automatically considered equal to true.
This is the standard behavior of JavaScript and JSX, which makes the code more concise.

How it works

When you write:

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

This is equivalent to the following:

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

Component **Composition**

— Component composition

Component Composition

— is a technique in which components are combined with each other to create more complex user interfaces

We create a Modal component that can be reused with different content inside
(The content will depend on prop.children)

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

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

  return (
    <div className="App">
      <button onClick={() => setIsOpen1(true)}> Open Modal Window 1 </button>
      <button onClick={() => setIsOpen2(true)}> Open Modal Window 2 </button>

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

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

2 different components, with their own unique content, for use in the “modal window” component

```jsx
function ModalContent1() {
  return (
    <div>
      <h2> Content of Modal Window 1 .</h2>
      <p> This is the content for the first case. </p>
    </div>
  );
}
```

```jsx
function ModalContent2() {
  return (
    <div>
      <h2> Content of Modal Window 2 !</h2>
      <p> This is the content for the second case. </p>
    </div>
  );
}
```

How to avoid props drilling

— with the help of props.children

How to avoid props drilling

— with the help of props.children

This approach allows avoiding passing data through intermediate components.
Instead, we simply pass the necessary components as children into NavBar.

```jsx
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>
  );
}
```

Explicit component passing

— instead of props.children

2 approaches:

Using props.children

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

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

Pros:

- Flexibility: Allows passing any number and types of child elements. This is convenient for components that act as containers or wrappers (for example, Card, Modal, Layout).
- Component composition: props.children is especially useful when you need to render any nested components inside another component.
- Clean code: Often results in more readable code when a component serves as a “container” for other components.

Cons:

- Implicitness: If you pass components through props.children, it is harder to control and understand what exactly is being passed inside, especially if components are passed from multiple places.
- Not suitable for specific cases: If strict control over the number or type of child components is required, using props.children may be less convenient.

Using explicit props to pass components

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

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

Pros:

- Explicitness: Explicitly passing components through props makes your component's interface more predictable and easy to read. You know exactly what and where will be rendered.
- Better control: Easier to control the number and types of components being passed.
- Simplifies typing: In TypeScript and PropTypes it is easier to describe exactly what is expected in the props.

Cons:

- Less flexibility: If you need to pass an arbitrary number of components or create complex compositions, explicit props may be too limiting.
- Redundancy: When a component accepts many components as props, the code can become bulky and harder to read.

useEffect

— Hook

useEffect

— is used to manage side effects in functional components.

useEffect

It allows executing code that should run after the component render, as well as managing side effects that occur due to state or prop changes.

useEffect

— How does it work?

useEffect

useEffect runs after every render of the component.
This means it works both on mounting (the first render) and on updating the component (re-render).

You can pass a dependency array as the second argument to useEffect.
If the dependency array is empty ([]), useEffect runs only when the component is mounted.

If the dependency array includes specific values, useEffect will run only when those values change.

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

  useEffect(() => {
    // Data fetching — side effect
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));

    // If a cleanup effect is needed (for example, unsubscribing from events):
    return () => {
      console.log('Cleanup before unmounting');
    };
  }, []);  // Empty dependency array — runs only once on mount

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

Unmounting: Unmounting: The function returned from useEffect is used to clean up side effects (for example, unsubscribing from events or stopping timers).

useEffect

— dependency array and more details about the process of how it works

useEffect

— Dependency array

Dependency array in useEffect

Defines in which exact case the code from useEffect should run.

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

  fetchData();
}, []);   // Empty dependency array – effect will run only on the first render
```

Variables in the dependency array act as a kind of event handlers to trigger the code inside useEffect.

If the dependency array is empty, the code inside useEffect will run only once — after the first render of the component and its rendering on the screen. If the array includes various states, then useEffect will run both after the initial render and after every update of those states.

1. First rendering:

• The component is rendered for the first time, React executes all the code of the component, including JSX calculations, reading states through useState, etc.
At this stage, useEffect does not run.

• After finishing the first render, React displays the result on the screen.

2. Execution of useEffect:

• After the render has finished and the changes are displayed on the screen, React starts executing the effects specified in useEffect.
If the effect has asynchronous code, for example, an API request, that request is sent, and the effect continues executing.

• The data received from the API is processed, and the component state is updated through setState.

3. Re-rendering after state change:

• When the state changes using setState, React triggers a re-render of the component.

• During the re-render, React again executes the entire component function, including JSX calculations.
However, effects (useEffect) at this stage do not run again if useEffect has an empty dependency array.

4. Re-execution of useEffect (when states are included in the dependency array):

• After the second render finishes, React again executes the effects, but only if the dependencies specified in the useEffect dependency array have changed.

Side Effects

— Side effects

Side Effect

— (side effects) are any operations that occur outside the current execution context of a function or component.

Actions that change external state, interact with external systems, or have consequences beyond the scope of the function.

— You can copy the code from the comments below and test it

Side Effects?

-are any actions that go beyond the current function and interact with the external world or modify state that exists outside the scope of this function.
In the context of React, side effects include:

- Fetching data from an API.
- Changing the page title.
- Subscribing to events such as window resize.
- Setting timers or intervals.
- Reading and writing to local storage (localStorage or sessionStorage).
- Getting the current user geolocation.
- Displaying system notifications.
- Connecting to a WebSocket to receive real-time data.
- Modifying global variables or states available outside the function.

Unsubscribing from events

We unsubscribe from everything that triggers more than once

1. Timers and intervals
- Why: They continue to run after the component is removed.
- Solution: Use clearTimeout or clearInterval.

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

2. Event listeners
- Why: Event handlers added via addEventListener remain active after the component is removed.
- Solution: Use removeEventListener.

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

3. WebSocket
- Why: The connection remains open and continues transmitting data.
- Solution: Close the connection.

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

4. Data subscriptions
- Why: Subscriptions (for example, RxJS) continue to receive data after the component is removed.
- Solution: Cancel the subscription.

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

5. Animations
- Why: Requests via requestAnimationFrame continue.
- Solution: Use cancelAnimationFrame.

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

6. Geolocation
- Why: Tracking via watchPosition continues.
- Solution: Use clearWatch.

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

7. Asynchronous operations
- Why: Attempting to update state after an asynchronous operation completes will cause an error.
- Solution: Use a flag for cancellation.

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

8. External APIs or third-party libraries
- Why: They may continue to use resources or data.
- Solution: Follow the documentation to cancel operations.

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

9. Browser events
- Why: Scroll, keyboard, or other global events remain active.
- Solution: Remove handlers.

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

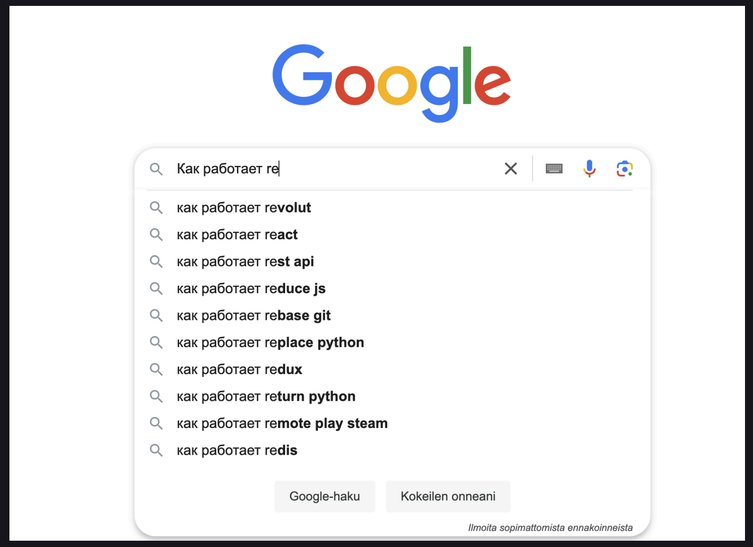
Race conditions

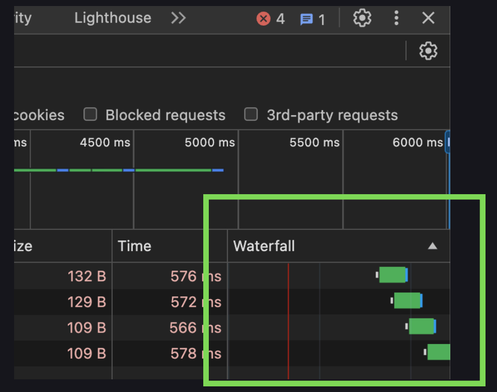
— Race conditions

race conditions

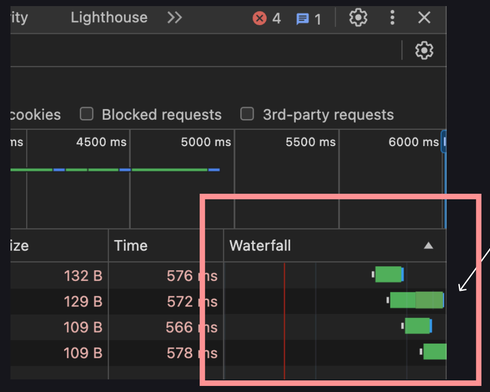
— race conditions

After each key press, a new request for data is created, and the data on the screen is updated
race conditions





If at some letter the request takes longer than the last one, then that stuck request will arrive last.
And, accordingly, its result will appear on the screen instead of the one we typed last.

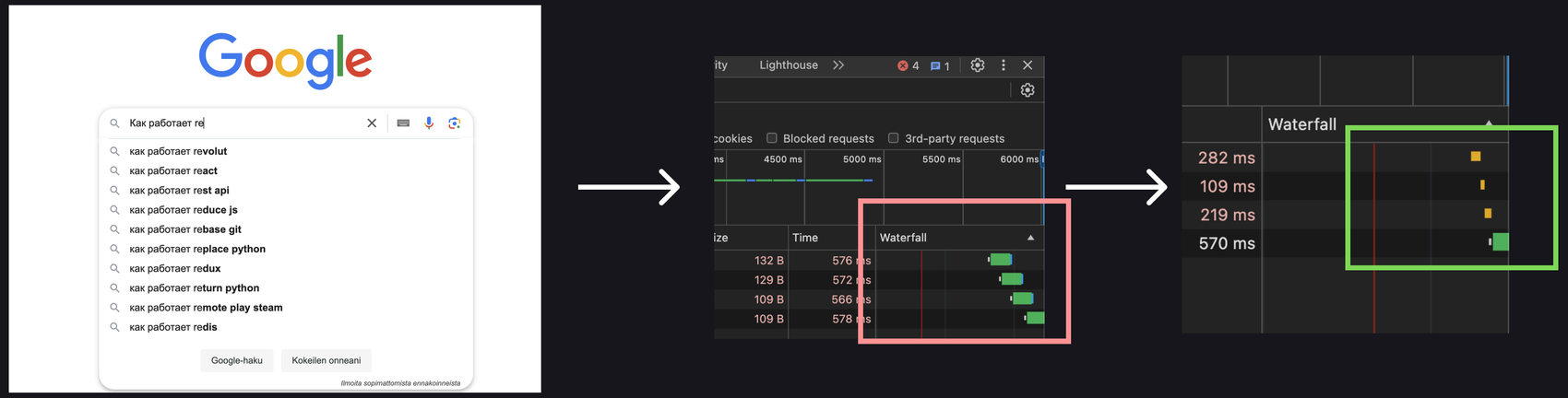


race conditions

— race conditions

— You can copy the code from the comments below and test it

To make everything work correctly, when creating a new request we first need to terminate the previous one.



AbortController will help us with this.

AbortController creates an object that can be used to cancel one or more asynchronous operations. It is used together with fetch to abort a request if there is no longer a need to execute it (for example, if the user switched to another screen or if the request was replaced by another one).

```
function MyComponent() {
  useEffect(() => {
    const controller = new AbortController()  // Create a new AbortController
    const signal = controller.signal;   // Get the signal from the controller

    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( 'The request was canceled' );
        } else {
          console.error(   'An error occurred:'    , err);
        }
      }
    }

    fetchData();

    return () => controller.abort();   // Abort the request on unmount or re-render
  }, []);  // Empty dependency array – effect runs only once on mount
}
```

REACT JS

race conditions

— more details

— You can copy the code from the comments below and test it.

AbortController is a built-in JavaScript API object.

1. **Creating an AbortController:** Inside useEffect, a new instance of AbortController is created, which provides a signal to control request cancellation.

2. **Passing signal to fetch:** The signal is passed in the fetch options, allowing the request to be canceled if AbortController.abort() is called.

3. **Canceling the previous request:** useEffect returns a cleanup function that calls controller.abort(). This cancels any previous request that might still be in progress if query changes before the request completes.

4. **Handling the cancellation error:** If the request is canceled, an error of type AbortError is thrown. We check for it in the catch block so that we don't show an error message to the user when the request was intentionally canceled.

```
function MyComponent() {
  useEffect(() => {
    const controller = new AbortController()  // Create a new AbortController
    const signal = controller.signal;    // Get the signal from the controller

    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( 'The request was canceled' );
        } else {
          console.error(    'An error occurred:'    , err);
        }
      }
    }

    fetchData();

    return () => controller.abort();    // Abort the request on unmount or re-render
  }, []);  // Empty dependency array – effect runs only once on mount
}
```

useEffect vs Events

— What to choose?

useEffect vs Events

— What to choose for working with side effects?

Use useEffect when you need automation.

Use useEffect when a side effect (for example, a server request) should happen automatically under certain conditions of the program.

For example, at specific time intervals or when certain states change. The moment when the side effect (for example, a server request) should occur is defined in the dependency array.

Also, useEffect allows running a function when the component is unmounted (removed).

Events with side effects

Use in 3 cases:

1. When executing a side effect (for example, a server request) is needed only after a user-triggered event (onClick, onChange, and so on).

2. When there is no need to run a side effect (for example, a server request) on the first load of the application (when the component is mounted).

3. When there is no need to run a cleanup function.

Class vs functional components

— and their differences

Class vs Function

— Class and functional components are two ways to create components in React

— You can copy the code from the comments below and test it

Class components

Class components in React were the main way of creating components before the introduction of hooks in React version 16.8 (January 2019). Before that, functional components were used only for simple tasks since they could not have state or lifecycle methods.

```
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;
```

Main features of class components:

• State: In class components, state is stored in the state object, which can be changed using the setState method.

• Lifecycle methods: Class components use special methods such as componentDidMount, componentDidUpdate, componentWillUnmount to perform actions at different stages of the component's lifecycle.

• Instances: For each class component, React creates an instance of the class, which stores state and methods.

Class vs Function

— Class and functional components are two ways to create components in React

— You can copy the code from the comments below and test it

Functional components

Functional components in React are a simpler way to create components. Initially, they were “stateless,” meaning they could not manage state and had no lifecycle methods. However, with the introduction of hooks in React version 16.8, functional components became more powerful and can now perform all the same tasks as class components.

```jsx
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;
```

Main features of functional components:

• Simplicity: Functional components are simpler in syntax and readability. They are just regular functions that take props and return JSX.

• Hooks: Functional components use hooks such as useState, useEffect, useContext to manage state and perform side effects.

• Absence of lifecycle methods: Functional components do not have lifecycle methods like class components. Instead, hooks such as useEffect perform similar tasks.

Class vs Function

— Class and functional components are two ways to create components in React

Functional components with the introduction of hooks have become the preferred way of creating components in React. They offer a simpler and more understandable syntax, and also allow easy management of state and side effects.

However, class components are still relevant and can be used in older projects or in cases where they are preferred.

Main differences

Class Components	Functional Components
Use ES6 class syntax	Use function syntax
Manage state with this.state	Manage state with hooks (useState)
Use lifecycle methods	Use hooks (useEffect)
Class instances are created on render	No instances (a component is just a function)
More complex and verbose syntax	Simpler and more concise syntax

Components, instances, elements

— and their differences

Components

A component is a fundamental part of a React application.
Components are functions that take input data (props) and return React elements describing how a part of the user interface should look.

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

Components are just templates that describe how a specific part of the interface should look.

Instances

Component instances refer to class components and are created when such components are rendered.
Each time React renders a class component, it creates a new instance of that class.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return <h1>Count: {this.state.count}</h1>;
  }
}
```

On each render, a new instance of the MyComponent component is created, which stores its state (this.state) and lifecycle methods.

Important: Functional components in React do not have instances, because they are not classes.

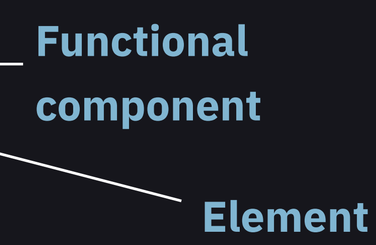
BUT

For simplicity and general understanding, many call the invocation of a functional component an instance.

Elements

An element is the simplest unit of React. Elements are a description of what should be rendered on the screen. They do not contain logic or state like components, and they do not have lifecycle methods. React elements are immutable objects.

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



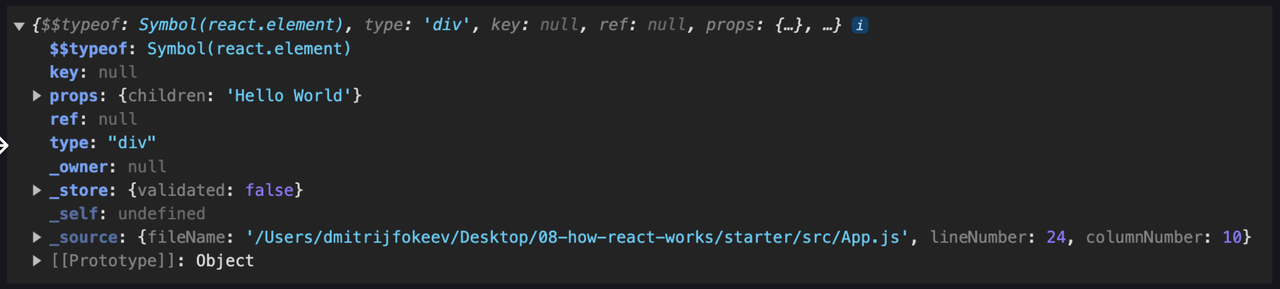
Simply put

An element is what a component returns.

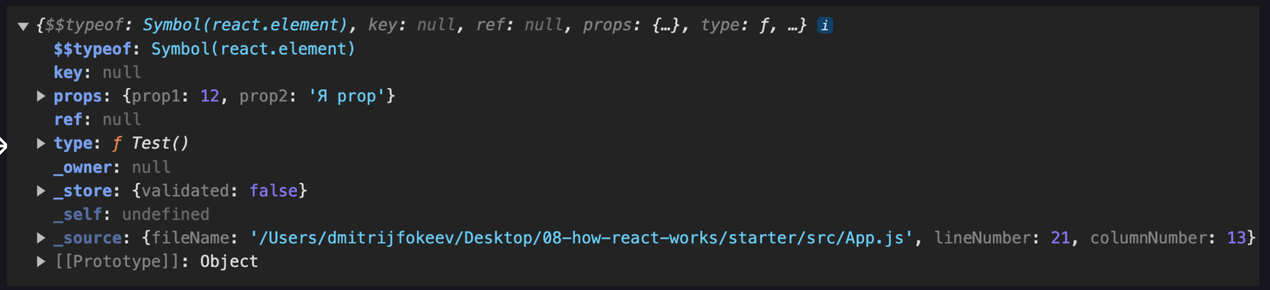
Element

An element in React is an object that React uses to describe what needs to be rendered in the DOM.
When you write JSX, such as Hello, world! , this JSX is compiled into a call to the React.createElement function, which creates an element object.

```
console.log(Test());

function Test() {
  return <div>Hello World</div>;
}
```




```
console.log(<Test prop1={12} prop2={"Я prop"} />);

function Test() {
  return <div>Hello World</div>;
}
```



Main differences in the top 2 calls:

- HTML tag vs. Component: In the first case, the element represents a standard HTML tag ('div'), and in the second case — a React component (Test).
- Rendering: In the first case, React renders the HTML tag, in the second — it calls the Test component to get the result of its rendering.

Rendering

— How it works

Component rendering in React

— is the process where React calls the component function to get a description of the interface in the form of React elements.
These elements are used to create and update the virtual DOM, which is then synchronized with the real DOM.

Rendering is not drawing on the screen, but drawing under the hood of React.

Internal rendering: When we talk about rendering in React, we often mean the process of calling the component, which returns React elements (via JSX). These elements form the virtual DOM — the internal representation of the interface that React works with.

Drawing: After rendering, React compares the new virtual DOM with the previous one to determine the minimal changes that need to be applied to the real DOM (this process is called “diffing”). Only after that are the changes synchronized with the real DOM, which leads to visible updates on the screen.

Rendering includes:

1. Calling the component and getting elements.
2. Building/updating the virtual DOM.
3. Comparing (diffing) the virtual DOM with the previous version.

And the actual drawing on the screen (synchronization with the real DOM) is the next step that React performs after rendering. Technically, it is called the Commit phase.

Virtual DOM

— What does it look like?

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

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

```jsx
{
  type: "div",
  props: {
    className: "app",
    children: [
      {
        type: SomeComponent,
        props: {}
      },
      showModal
        ? {
            type: ModalWindow,
            props: {
              children: {
                type: Overlay,
                props: {
                  children: [
                    {
                      type: "h3",
                      props: {
                        children: "Leave a review"
                      }
                    },
                    {
                      type: "button",
                      props: {
                        children: "5 stars!"
                      }
                    }
                  ]
                }
              }
            }
          }
        : null,
      {
        type: Btn,
        props: {
          children: showModal ? "Rate" : "Close window"
        }
      }
    ]
  }
}
```

Rendering

— When is rendering triggered?

Component rendering is triggered in two cases:

Initial rendering:
Happens when a component is first added to the DOM.
At this moment React calls the component to get its JSX (or element), and then builds the virtual DOM, which is synchronized with the real DOM.

Rerender:
Happens every time the props or state of the component change.
React calls the component again, compares the new virtual DOM with the previous one (this is called “diffing”), and updates the real DOM only where changes actually occurred (if changes really happened).

Virtual DOM

— How it works?

Virtual DOM

— it is an object containing a tree of components and elements.

Virtual DOM

- is a lightweight, abstract copy of the real DOM that exists in memory and is used to manage the interface.
The Virtual DOM is a tree of objects where each object corresponds to a UI node, such as an HTML element or a React component.

How does the Virtual DOM work?

1. Creating the Virtual DOM:

• When a component is rendered for the first time, React creates the Virtual DOM — a description of the interface in the form of a tree of elements.
• This tree is created based on the JSX returned by the component and serves as the input for building the Fiber Tree structure.

2. Updating and comparing (diffing):

• When state or props change, React creates a new Virtual DOM tree.
• This tree is used to update the “work-in-progress” Fiber Tree, which is compared with the current state (current Fiber Tree).
• React identifies minimal changes at the Fiber Tree level, not through direct Virtual DOM comparison.

3. Applying changes:

• After computing the changes at the Fiber Tree level, React proceeds to the “Commit” phase.
• At this stage, the changes are applied to the real DOM: only the elements that changed are updated.
• Side effects such as useEffect or componentDidUpdate are also triggered.

Fiber tree

— How it works?

Fiber tree

— it is not just a static structure, but a dynamic and flexible system that:

1. Links the virtual DOM with the real DOM.

2. Stores the state of the application (including hooks and effects).

3. Allows React to efficiently manage rendering:

• Tracks changes in components.
• Manages tasks with different priorities.
• Optimizes rerendering through an incremental approach.

Fiber Tree — is it an object?

Fiber is an object that React creates for each element of the virtual DOM. This object contains all the necessary information to describe the element and manage its lifecycle.

Each Fiber object contains many properties. Here are the main ones:

1. Type of element:
• Specifies which type of component is associated with this Fiber:
• String ('div', 'span', etc.) for HTML elements.
• Function or class for React components.
• null for fragments (React.Fragment).
2. Key:
• Used to identify nodes in lists and helps React track changes.
3. Props:
• Store the properties passed to the element or component.
4. State:
• Contains the internal state of the component (if it is a stateful component).
5. Parent, child, and sibling nodes:
• return: Reference to the parent Fiber.
• child: Reference to the first child Fiber.
• sibling: Reference to the next node on the same level.
6. Effect List:
• A list of side effects (for example, hook calls like useEffect, DOM changes) associated with this Fiber.
7. Reference to the DOM node:
• Contains a reference to the corresponding real DOM node (for example,), if it has already been created.
8. Update priority:
• Used to manage the order of task execution (for example, high-priority updates for animations).
9. Lifecycle algorithm:
• Information about the current state of the component: for example, whether a rerender is needed, whether there are effects, etc.

Fiber tree

— this is the basis of the new React architecture, introduced in React 16 (2017), which enables component rendering.

How it worked before (pre-Fiber):

•Before Fiber appeared, React worked synchronously. When state or props changed, React fully updated the virtual DOM and calculated all the changes that needed to be applied to the real DOM. This process could not be interrupted.

• For example, if rendering took 5 seconds, the entire application was blocked for that time.
User interactions such as clicks, key presses, or scrolling could not be processed until rendering was complete.

How it works now with Fiber:

• In the new Fiber architecture, React can assign priorities to different tasks. For example, processing user interactions (clicks, text input) has a higher priority than rendering less important parts of the interface.

• If rendering takes a long time, and during this time something more important happens (for example, the user clicks a button), React can pause the current rendering, execute the higher-priority task (for example, handle the click), and then return to the unfinished rendering.

• React can split rendering into small pieces, execute them gradually, and check in between whether there are more prioritized tasks.

Example:

1. Before Fiber:
The user clicks a button, which triggers the update of a large list of elements. Rendering this list takes 5 seconds. During these 5 seconds, the interface does not respond to other actions. If the user tries to click another button, it won't work until the list rendering is finished.

2. With Fiber:
The user clicks the same button, and rendering of the list begins. After 2 seconds, the user clicks another button. Thanks to Fiber, React pauses the list rendering, handles the click on the second button (the higher-priority action), and only after that returns to finishing the list rendering.

Fiber tree vs **Virtual DOM**

— What’s the difference

Life analogy

Renovation in the house

Imagine you are renovating an apartment.

The house as an interface:
Your real house (the physical apartment) is the real DOM.
You want to make changes in the house: repaint the walls, move the furniture, or replace the wallpaper.

Roles of Virtual DOM and Fiber Tree:

Virtual DOM — is the renovation plan:
First, you make a plan — on paper or in a computer program.
You draw how your apartment will look after the renovation (for example, new furniture, wallpaper, wall colors).
This is the analogue of the Virtual DOM, which describes how the interface should look.

2. Fiber Tree — is the renovation manager:
You need a person who will oversee the implementation of your plan, evaluate what already exists in the apartment, and minimize the amount of work.

For example:
If the furniture is already in place, it doesn't need to be moved.
If the wall is already painted in the desired color, it doesn't need to be repainted.

This "renovation manager" is the Fiber Tree.
It helps decide what needs to be done first, what can be postponed, and which tasks don't need to be done at all.

Life analogy

How everything works together:

1. Creating the plan (Virtual DOM):
• You create a new version of your plan (for example, add a new sofa in the living room).
• The Virtual DOM describes all the elements and changes that should be made in the apartment.

2. Comparing the old and new plan (Diffing):
You compare the old version of the plan (what was) with the new one (what you want).

For example:
You see that the table in the kitchen is in the same place — it doesn't need to be touched.
But in the living room, the old sofa needs to be removed and the new one put in.

3. Managing the process (Fiber Tree):
The Fiber Tree, like a manager, decides exactly what needs to be done and in what order:
• First remove the old sofa.
• Then put in the new one.
• Finally, repaint the walls.
The Fiber Tree also ensures that you don't redo work twice and that changes are made only where needed.

4. Executing the renovation (Real DOM):
The manager (Fiber Tree) sends the workers to make changes in your apartment.
Similarly, React updates the real DOM only where it is truly necessary.

Main difference:

• The Virtual DOM is just the plan (what you want to change). By itself, it doesn't know what's already done or in what order to act.
• The Fiber Tree is the manager who compares the current situation with the plan, sets priorities, and ensures that the work is carried out as efficiently as possible.

Comparison table

Differences between Virtual DOM and Fiber Tree:

Criterion	Virtual DOM	Fiber Tree
Purpose	Lightweight representation of the interface structure.	Manages rendering, state, and effects.
Form	Simple tree structure of JavaScript objects.	Extended tree structure of Fiber objects.
Data	Only information about interface elements.	Stores state, hooks, DOM references, etc.
DOM Management	Used to calculate changes.	Directly connects to the real DOM.
Rendering	Determines what needs to be updated.	Manages the update process and priorities.
Incrementality	No (one-time tree comparison).	Yes (supports interruptible rendering).
Priorities	Does not manage priorities.	Supports task prioritization.

Rendering scheme

— How React updates the application

Rendering scheme

1

Start of the rendering process

Initialization of the program a function call like setState or receiving new props in a component signals React that the interface needs to be updated.

2

Creating a new Virtual DOM

Components return JSX, which is transformed into a new Virtual DOM tree.

3

Building and updating the Fiber Tree

Based on the Virtual DOM, React creates a new Fiber Tree. Each node in the Fiber Tree corresponds to a node in the Virtual DOM and stores information about the component, its state, props, and links to parent, child, and sibling nodes.

The core (Fiber mechanism) splits rendering into small tasks (units of work), which allows React to perform rendering asynchronously and interrupt it if necessary.

4

Reconciliation (Diffing)

React uses the Fiber mechanism to compare the new Virtual DOM with the previous version. This is called diffing.
This way React understands which parts of the real DOM need to be updated.

Fiber helps React decide which nodes of the Virtual DOM can be reused and which need to be updated or removed.
This process allows React to minimize changes to the real DOM.

5

Committing changes to the real DOM (Commit Phase)

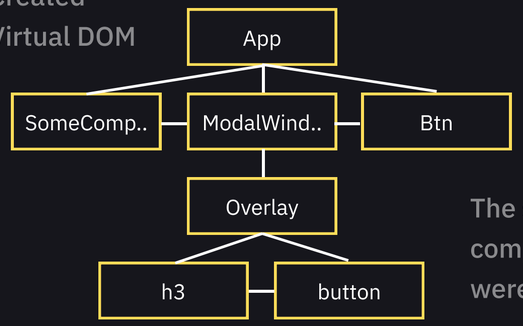
After comparison, React prepares the changes that need to be applied to the real DOM. This stage is called the commit phase.

React applies the changes to the real DOM based on its calculations.
The Fiber core controls the process to ensure that only the necessary changes are made to the DOM.

Example

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

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

Created Virtual DOM



ShowModal(true)

useState

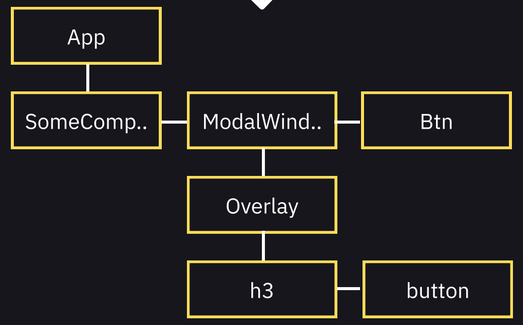
New Virtual DOM

setShowModal(false)

The state was updated in the App component, so all child components were updated.

Fiber Tree is created

Fiber Tree



The Fiber core manages the comparison process.

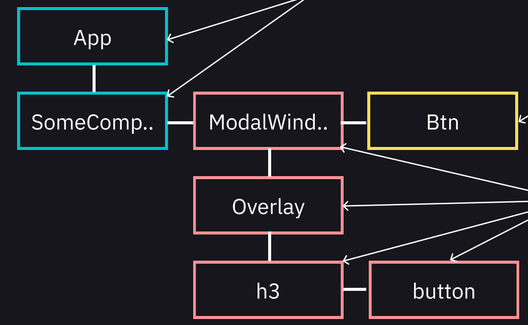
diffing & Reconciliation Comparison of the old Virtual DOM and the updated one

Final decision of the Fiber mechanism about the update

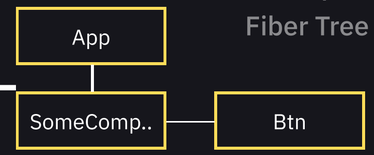
Leave as is

Update text

Delete



Final updated Fiber Tree



Rendering to t he page

Fiber Tree is not just a copy of the Virtual DOM, but a structure that stores additional data for optimization, for example:

- References to child and parent nodes.
- Update priorities (for example, for animations or user interactions).
- Hooks and their state (for example, useState and useEffect).

Memoization

— Application performance optimization

Re-render

— Re-rendering of child components always happens if the parent component re-renders, even if the child component’s dependencies received from the parent (props) have not changed.

— You can copy the code from the comments below and test it

Such an approach often wastes extra resources on re-drawing and re-rendering.

Test the code from the comments.

1 — Increase count by pressing the button
2 — Show articles by pressing the button
3 — Increase count by pressing the button

You will see a noticeable delay.

Memoization to the rescue

Memoization

Memoization is an optimization technique that consists of caching the results of computations.
If the input data doesn’t change, the computation is not repeated — the cached result is returned.

In React, memoization is used to prevent repeated computations and re-renders if the input data (dependencies) have not changed.
This is achieved through useMemo, useCallback, and memo.

memo()

— this is a higher-order function (Higher-Order Component, HOC) that prevents a component from re-rendering if its props have not changed.

— You can copy the code from the comments below and test it

Solution to the problem from the previous slide:

Test the code from the comments.

1 — Increase count by pressing the button
2 — Show articles by pressing the button
3 — Increase count by pressing the button

You will see that the problem has disappeared.

```
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 compares the props of the Archive component (in this case, showPosts). If the props have not changed, memo prevents the component from rendering. This helps improve performance since the component does not need to be re-rendered if the props remain the same.

When to use?
Use memo for functional components that re-render too often but don’t actually need to if the props remain unchanged.

Commite **Phase**

— Commite Phase

Commite Phase

— The final stage of rendering.

Commit Phase

— In this phase React applies all the changes calculated during the Render Phase to the real DOM.

This phase includes:

Updating the real DOM: Applying changes to the DOM structure, such as adding, removing, or modifying elements.

Executing effects: For components using useEffect, effects marked to run after rendering are also executed in this phase.

Unlike the Render Phase, which can be paused, the Commit Phase is always executed synchronously.
This ensures that the interface is fully updated and all necessary changes are applied in real time.

ReactDOM

— The real rendering in the browser is not performed by the React library, but by the ReactDOM library.

React & ReactDOM

The actual rendering in the browser is done by another library — ReactDOM.

src > JS index.js > ...

```
1  import React from "react";
2  import ReactDOM from "react-dom/client";
```

Such libraries as ReactDOM are called RENDERERS.

Although we now know that the real rendering happens under the hood, these libraries simply draw the result.

Other RENDERER libraries for React:

The most obvious example is React Native

React Native — used for rendering mobile applications on iOS and Android platforms. Instead of working with the browser DOM, React Native transforms React components into native platform elements such as View, Text, and Image.
React 360 — for creating virtual and augmented reality experiences.
Ink allows rendering React components into the console or terminal.
React PDF used for generating PDF documents using React components.

...And many more.

Diffing

— How it works

Diffing

— This is the algorithm React uses to compare the old and the new Virtual DOM. Diffing determines which parts of the interface have changed and calculates the minimal updates needed for the real DOM.

2 rules by which diffing comparison works:

First: Comparison by element/component types at the same nesting level.

If elements at the same level have the same type (for example, and), React assumes it is the same element and compares their attributes and content. If the types match, React does a shallow compare of the attributes and updates only those that have changed.

If the node types differ (for example, and), React removes the old element and creates a new one, since it assumes these nodes represent different elements.

Second: Comparison of lists (keyed diffing).

If React encounters a list of elements, it uses special keys (key) to track and compare elements. This helps React accurately determine which elements were added, removed, or moved.

Keys must be unique among siblings so that React can correctly track changes.

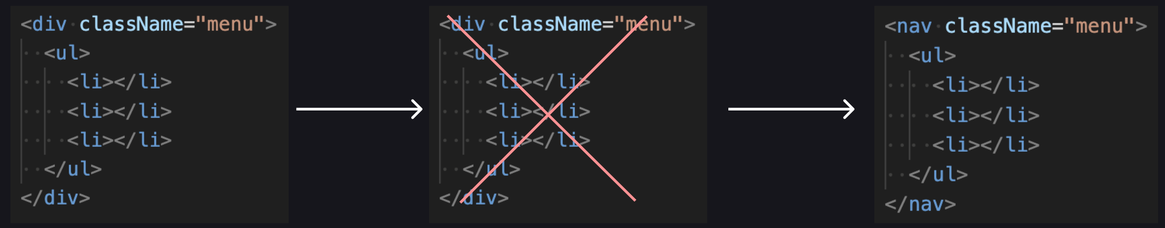
If React detects that the text inside a node has changed, it updates the text content in the real DOM without touching the element itself.

Diffing

— scheme

The same position, but different elements

That means React needs to recreate the element itself along with its child elements and reset all the state of those elements.



The same applies if it’s a component instead of an HTML element.

Diffing

— comparison of keys

— You can copy the code from the comments below and test it.

The same position, the same elements but without key

If in our program, when state changes, identical elements are created in identical places, this can lead to problems when rendering these identical elements. State will be the same for actually different elements (which are technically named the same and are located in the same place).

(Reminder: changing state or props triggers the rendering process, but during rendering the diffing mechanism is used, which decides which part of the program to update and which to leave unchanged for maximum efficiency).

Example with tabs:

```jsx
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. *Inside the “Test” component there is code for buttons.*
These buttons display one tab or another.

4-*That same unique Key that helps React understand that these are not the same element, but 3 different ones.*
(Remove it and increase the increment, you will see that each tab shows the same number).

3-*This is creating a tab based on which button is pressed. (In theory, there should be 3 buttons and 3 tabs for each button).*

2-*This is the component of the tab itself. It has a button that increases the state variable count by 1. The count variable is displayed in the tab content on the page. Changing this state triggers rendering.*

1-*Press to increase the number in the tab.*

2-*The rendering process starts.*

3-*Diffing compares the new Virtual DOM with the old one by two key things:*
- whether the types of elements and their positions have changed;
- by keys;
Also checks text:
- whether the text inside the element has changed. If yes, only it is updated.

4-*If in the <TabContent> component there is no key, then diffing does not update this element on the page, since it is the same and in the same place. React only updates the text, but the state remains.*

If an element has no key, and if the element has the same name and is in the same place as it was, its state will be preserved during re-rendering.

Batching updates of states

— State update batching

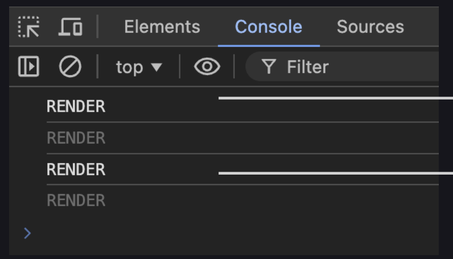
Batching

— updates of states

Batching updates of states

— is the process by which React combines multiple setState calls into a single render, in order to minimize the number of component re-renders and improve application performance.
In React 18, batching became smarter: now React automatically batches all state updates that happen within one event or asynchronous code, thus avoiding unnecessary re-renders.

```
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}>
        Increace by 3
      </button>
    </div>
  );
}
```



Render when the application starts

Render after pressing the button

Three state updates should trigger three renders.
But if we run this code, we will see the word “render” only once in the console.
(Render always starts with the execution of the component function.)

— You can copy the code from the comments below and test it

Asynchronous

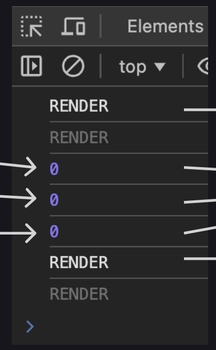
— state update due to batching updates

— You can copy the code from the comments below and test it.

State updates happen asynchronously.

Batching updates: React combines multiple setState calls into a single update to avoid redundant rendering and improve performance. This means that setState calls do not immediately update the component; instead, they are deferred and executed together in one render.

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

  function increaseNumByThree() {
    setNumber(number + 1);
    console.log(number);   // We expect it to be 1, but the console will show 0
    setNumber(number + 1);
    console.log(number);   // We expect it to be 2, but the console will show 0
    setNumber(number + 1);
    console.log(number);   // We expect it to be 3, but the console will show 0
  }

  return (
    <div>
      <p style={{ fontSize: "24px" }}>{number}</p>

      <button style={{ padding: "8px" }} onClick={increaseNumByThree}>
        Increace by 3
      </button>
    </div>
  );
}
```



Render when the application starts

After pressing the button

Render after pressing the button

Three state updates should trigger three renders.
But if we run this code, we will see the word “render” only once in the console.
(Render always starts with the execution of the component function.)

Question!?
What will actually be displayed on the screen after pressing the button? Number 3 or 1?

Answer:
Number 1

Because of the asynchronous nature of state updates (made for optimization), after pressing the button React will do the following:

1. Run the function increaseNumByThree().
2. React will collect all three state updates into a batch and apply them one by one, but it will apply them starting from zero (the old state) all three times.
3. Run one rendering and display the result.

Using the updated value

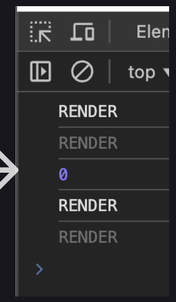
— how to update the current value and use it when necessary

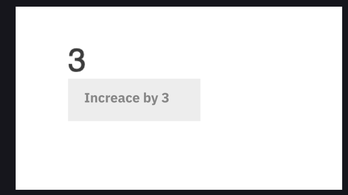
Rule!

When you need to update state based on its previous state, you must get the old state from the callback function of 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)  // Still the same 0. But on the page after render it's 3
    setNumber((number) => number + 1);
    setNumber((number) => number + 1);
  }

  return (
    <div>
      <p style={{ fontSize: "24px" }}>{number}</p>

      <button style={{ padding: "8px" }} onClick={increaseNumByThree}>
        Increace by 3
      </button>
    </div>
  );
}
```





— You can copy the code from the comments below and test it.

Mount & Unmount

— Components

Mount & Unmount

— Mounting and unmounting components

In React, components go through several lifecycle stages, and the important stages are mounting (mount) and unmounting (unmount).

1 Mounting
-The component is rendered for the first time
-States and props are created

2 Re-render (Optional) happens when:
-State changes
-Props change
-The parent component re-renders
-Context changes (more on this later)

→ **Re-render can happen many times**

3 Unmounting
-The component is removed
-States and props are deleted

Framework vs library

— and their differences

Library

— You control which specialists to use (which packages to use for solving certain tasks)

Analogy: Home renovation

Library: You control which specialists to use (which packages to use for solving certain tasks).

Imagine you are renovating your apartment, and you need different specialists: an electrician, a tiler, a painter, etc.

You decide yourself whom and when to call, and each specialist performs only their narrow task (for example, the tiler lays tiles, and the electrician installs wiring).
In this case, you control the entire renovation process, and the specialists just help carry out specific tasks.

Programming example:

Libraries (for example, React, lodash, jQuery) are a set of ready-made tools that you call when you need them.

Framework: You hire a company for a "turnkey renovation."

Imagine that you don't want to manage the renovation yourself, so you hire a construction company.
It takes over the whole process: planning, buying materials, coordinating specialists, and delivering the finished result.
The company dictates the rules of work: you can influence only certain details (for example, choose the wall color), but you do not control how and in what order everything is done.
In this case, the framework manages the process, and you only provide the necessary input.

Programming example:

• Frameworks (for example, Angular, Django) control the entire development process.
You follow their structure and only provide specific parts of the logic.

use**Ref**

— preserves data between renders

REACT JS

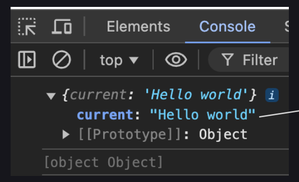
useRef()

— It is often used for working with DOM elements, storing values that remain unchanged between renders, and creating variables that should not trigger a component re-render.

useRef() what is this?

useRef() - is a hook for creating a variable that will be an object.
The value passed to useRef(value) is stored in the current property.

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



An object with a current property.

useRef() - preserves its value between renders, but does not trigger a re-render.

3 main use cases of useRef():

1. Storing DOM elements:
• useRef is often used to directly access DOM elements. This is useful when you need to manually control an element, for example, focusing on an input.
2. Storing mutable values:
• You can use useRef to store values that need to be changed without re-rendering the component. Unlike useState, changing a useRef value does not cause the component to re-render.
3. Preserving values between renders:
• The value stored in useRef remains unchanged between renders, which makes it ideal for storing values that should “survive” component updates.

THE DATA OF A VARIABLE DOES NOT CHANGE DURING THE PROGRAM EXECUTION

No → A REGULAR VARIABLE

Yes
they change ↓

SHOULD THE COMPONENT RE-RENDER AFTER THE DATA CHANGES?

No → USEREF

Yes
it should ↓

CREATE USESTATE →

NEW DATA OF THE USESTATE VARIABLE DEPENDS ON THE OLD DATA OF THIS USESTATE VARIABLE

No →

```
const [newNumber, setNumber] = useState(1);
<button onClick={setNewNumber(10)}></button>
```

Yes ↓

```
const [newNumber, setNumber] = useState(1);
<button onClick={setNewNumber((newNumber) => newNumber + 10)}></button>
```

The new value is obtained using the old one

useRef()

— for storing a DOM element

useRef() for storing a DOM element

1. We create a component.
2. We create a variable and assign it the value useRef(null).
3. The component returns an HTML element.
4. In the attributes of the HTML element, we write the ref prop with the value of the variable created using the useRef hook: ref={refElement}.

Thus, in the variable created with the useRef hook, there will always be stored a reference to this element.

```
function Input() {
  const refElement = useRef(null);

  useEffect(() => {
    refElement.current.focus();
  }, []);

  return <input ref={refElement} />;
}
```

Example:
To make an <input> element automatically focus after the page loads, we use the useEffect hook, which runs once after rendering.

(We use useEffect() because we need to first create the entire page and its elements, and only then do something with those elements).

custom Hook

— do it yourself

Custom hooks

— These are functions that allow reusing state logic and effects across multiple components.

Why do we need custom hooks?

1. Reusing logic:
• If you notice that the same logic is used in multiple components, you can extract it into a custom hook to avoid code duplication.

2. Code organization:
• Custom hooks help split complex components into smaller, more manageable parts, improving readability and maintainability of code.

3. Encapsulation of logic:
• Custom hooks allow encapsulating logic, hiding it from the main component, which makes the component simpler and clearer.

2 rules for creating custom hooks

1- A custom hook must use at least one standard hook (useState, useEffect, useRef, useContext, etc.).

2- The name of a custom hook function must start with the prefix “use”.

Example #1

— A custom hook that fetches data from an API

— You can copy the code from the comments below and test it

Custom Hook

```
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;
```

Using the hook in a component

```
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;
```

Example #2

— A custom hook that returns the width and height of the browser window

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

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

  useEffect(() => {
    // Window resize event handler
    function handleResize() {
      setWindowSize({
        width: window.innerWidth,
        height: window.innerHeight
      });
    }

    // Add event listener
    window.addEventListener('resize', handleResize);

    // Remove event listener when component unmounts
    return () => window.removeEventListener('resize', handleResize);
  }, []); // Empty dependency array means the effect runs only on mount

  return windowSize; // Return the current window size
}

// Using the custom hook in a component
export default function MyComponent() {
  const windowSize = useWindowSize();

  return (
    <div>
      <h1> Window size </h1>
      <p> Width: {windowSize.width}px</p>
      <p> Height: {windowSize.height}px</p>
    </div>
  );
}
```

— You can copy the code from the comments below and test it

Example #2

— Custom hook with parameters

— You can copy the code from the comments below and test it.

A custom hook can take arguments just like a regular function

```
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(); // Check on initial load
    window.addEventListener('resize', handleResize);

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

  return isWithinRange;
}

// Using the custom hook in a component
export default function MyComponent() {
  const isWithinRange = useWindowSize(500, 1200);

  return (
    <div>
      <h1> Window size </h1>
      {isWithinRange ? (
        <p> Window width is within the specified range. </p>
      ) : (
        <p> Window width is outside the specified range. </p>
      )}
    </div>
  );
}
```

React **Router**

— library

React Router 6.4+

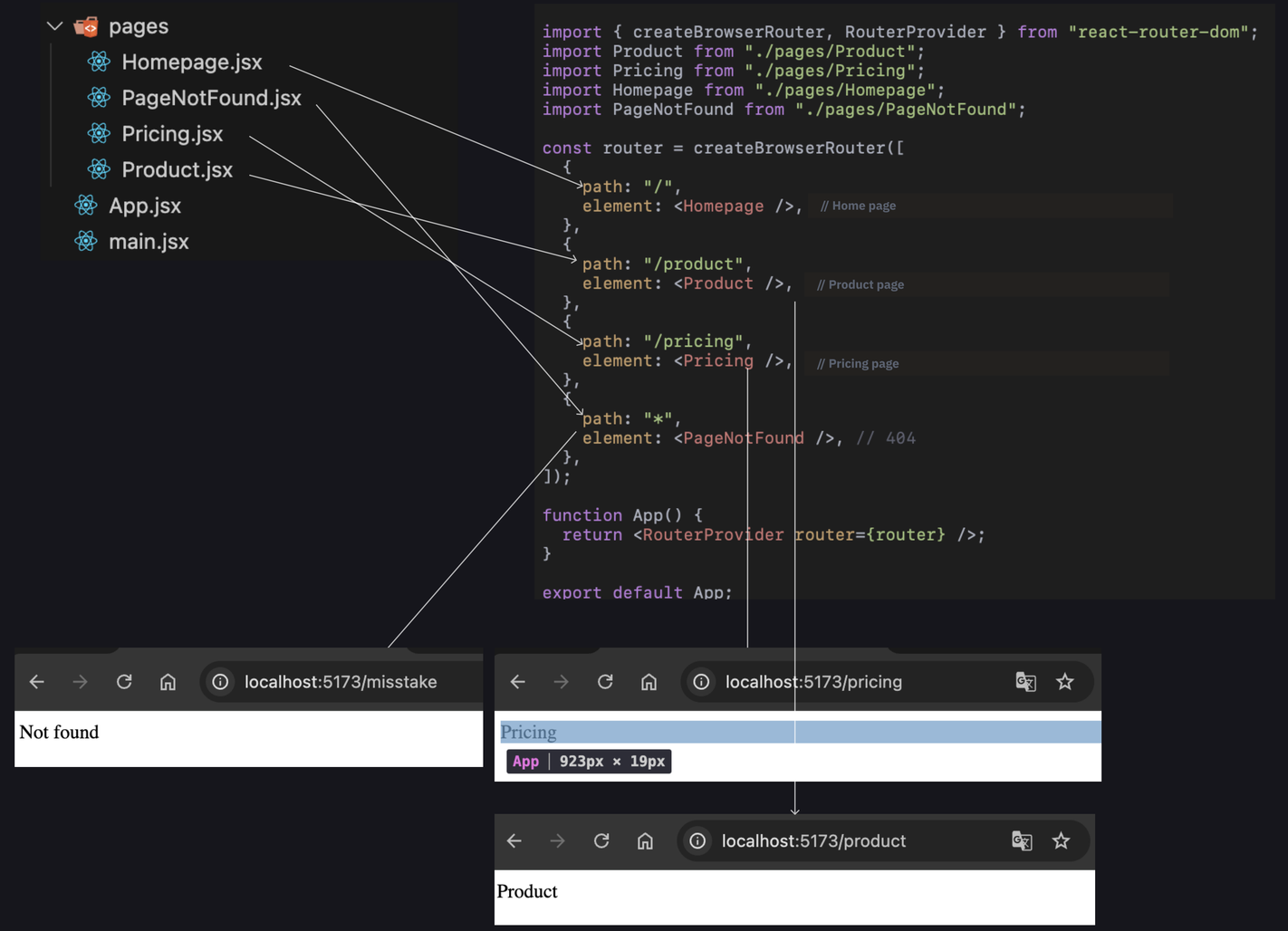
— This is a library for managing routing in React-based applications.
It allows developers to create single-page applications (SPA) with multiple “pages” or sections, while maintaining smoothness and speed without a full page reload.

Modern syntax

Installing a library into the project:

In the project folder, type in the terminal:

npm i react-router-dom



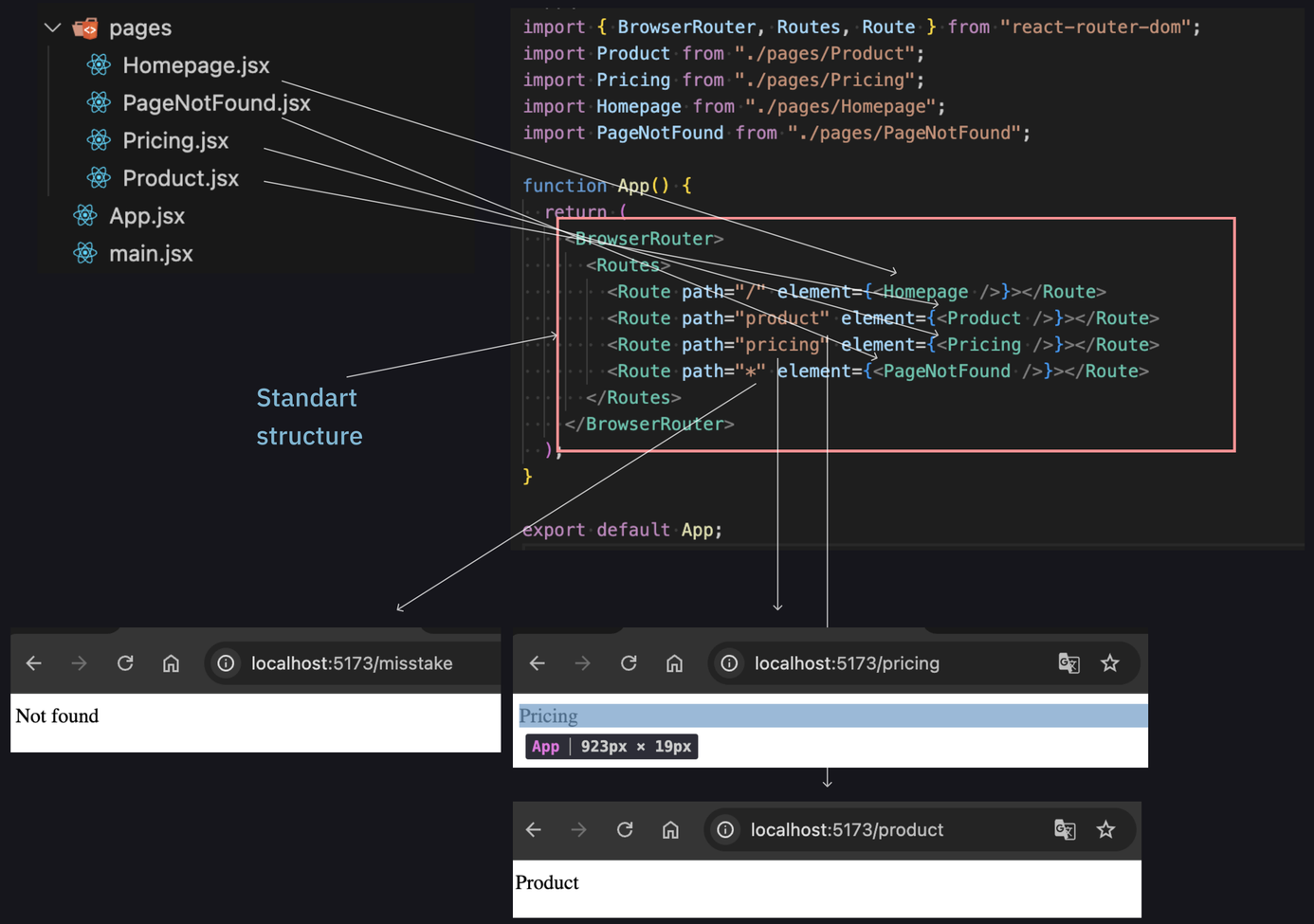
React Router before 6.4

Outdated syntax. Before version 6.4

Installing a library into the project:

In the project folder, type in the terminal:

npm i react-router-dom



Router **<Link>**

— how links work

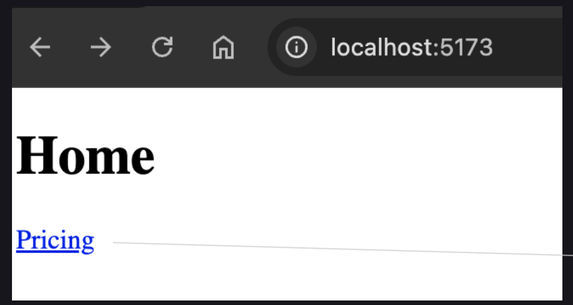
Links <Link>

— In React Router they work as navigation elements that allow users to move between different “pages” or sections of your application without reloading the page.

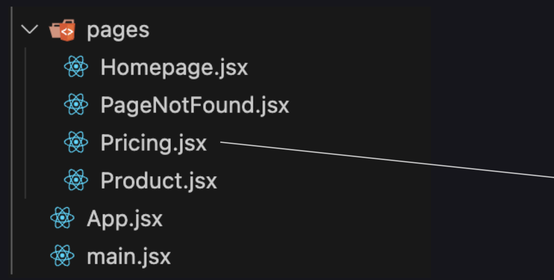
How links work in router:

The <Link> component in React Router is used to create links inside the application. The to attribute specifies the path to navigate to.

It can be a string (for example, /about) or an object that allows more flexible navigation settings.




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

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

export default Homepage;
```




```jsx
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>
  );
}
```

Links <Link>

— Additional features. Object instead of text

An object can include additional properties, such as:

- **pathname** — the path to the page.
- **search** — query string parameters (for example, ?sort=asc).
- **hash** — an anchor on the page (for example, #section1).
- **state** — data that can be passed into the route.

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

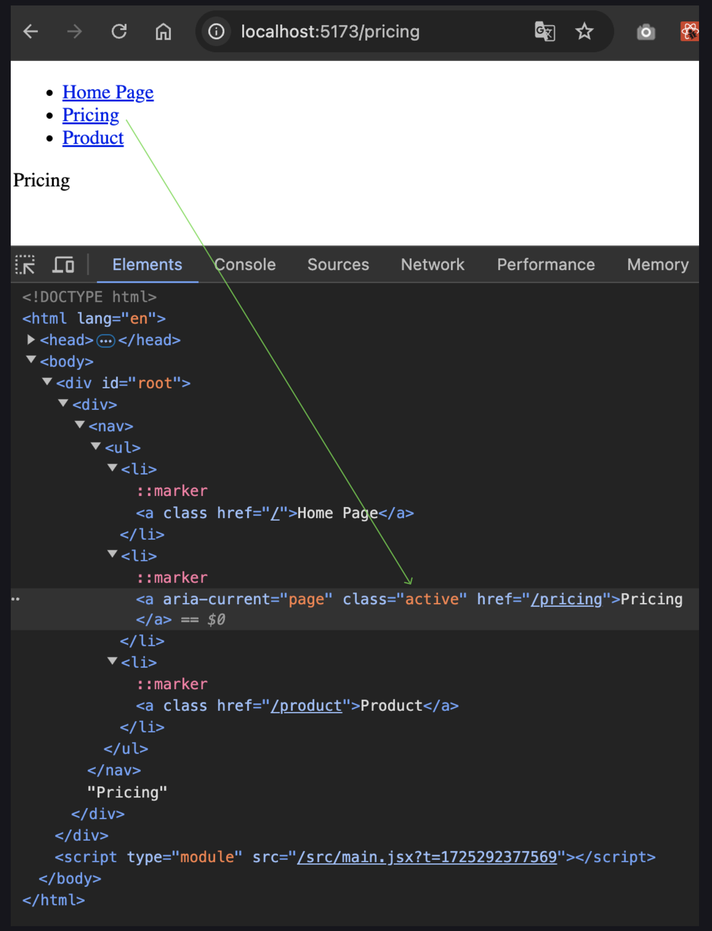
- **pathname: "/about"** — specifies the path to the page.
- **search: "?sort=asc"** — adds query parameters to the URL.
- **hash: "#info"** — scrolls the page to the element with the identifier info.
- **state: { fromHome: true }** — passes data (for example, fromHome), which can be used in the target component via the useLocation hook.

<NavLink> links

— This is a special component in React Router, which is an extension of <Link>.
It is used to create links with an active state (for example, highlighting the currently active link).

Example of <NavLink>:

An active class is added to the clicked link.




```jsx
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 />

— like {children} but only for pages

<Outlet />

— This is a component provided by React Router that is used in a route component to render child routes. It works like a “placeholder” in your component where the child route elements will be displayed.

Route structure:

• When you have routes that include child routes (nested routes), you can use <Outlet /> in the parent component to specify where those child routes should be displayed.
• The parent route renders its component, and if there is an <Outlet /> inside that component, then the corresponding child route will be rendered in that place.

```
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;
```

Dashboard is the parent route, and Settings is the child. To render <Settings /> inside <Dashboard />, you add <Outlet /> in the Dashboard component:

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

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

export default Dashboard;
```

How <Outlet /> works:
• When the user goes to /dashboard, the Dashboard component is displayed.
• If the user goes to /dashboard/settings, then the Settings component is rendered inside the Dashboard component, right where the <Outlet /> is located.

Router, the **index** attribute

— the default child page

Router, index attribute

— is used to specify which route should be displayed by default if the parent route matches the URL but no specific child route is given.

How index works:

Main idea:

• Instead of creating a separate route with a specific path, you create a route with the index attribute, which will be rendered if the path matches the parent but no specific child route is provided.

```
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 /> },
    ],
  },
]);
```

• The parent route /dashboard renders the Dashboard component.
• If the user goes to /dashboard but doesn't specify any child route, React Router automatically renders the component associated with the index route, in this case, Overview.
• If the user goes to /dashboard/settings, then the Settings component is rendered.

Router **useParams()**

— Hook. Gets the current route parameter

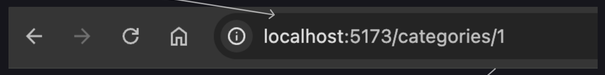
useParams()

— This is a hook from the React Router library that allows you to get the parameters of the current route.
It returns an object where the keys are the names of the parameters defined in your route, and the values are the current values of those parameters.

How it works:

1. Defining a route with parameters:
Route parameters are defined with a colon (:) before the parameter name in the route path. Here :id is a route parameter.

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



Property name

Property value

2. Using useParams() in a component:
When the route is active (i.e., the current URL matches the route defined in Route), React Router will extract the parameter value from the URL and pass it through useParams().
Example of use:

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

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

export default Categories;
```

3. Getting parameters:
If the current URL is, for example, /cities/123, then calling useParams() will return the object:

Router **useSearchParams()**

— Hook. Get, modify, and add query string

useSearchParams()

— It is a hook that provides the ability to work with query parameters in the URL in React applications that use the react-router-dom library.

With it you can get, modify, and add query parameters in the URL without reloading the page.

What is a query string parameter:

– It's what comes after the ? in a URL. For example:
http://example.com/?name=John
Here name is the parameter name, and John is its value.

You can programmatically update the URL by adding or changing parameters, for example, from ?name=John to ?name=Jane. The page will not reload, but your application will detect the change and can react to it (for example, filter a list or display different data).

When is this used?

Imagine you have a product listing page. When clicking the filter "Show only sale items," you want to change the URL so it includes the parameter ?sale=true. Then, if the user saves this link or reloads the page, the filter will still be applied.

How it works:

When you call useSearchParams(), you get a tuple of two values:

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

searchParams — an object to read current parameters from the address bar.
setSearchParams — a function to modify parameters.

Getting a parameter:
searchParams.get('name') returns the value of the 'name' parameter, or null if it doesn't exist.

Modifying parameters:
• setSearchParams({ name: value }) sets the URL to ?name=..., where ... is your value

useSearchParams()

— This is a hook that provides the ability to work with query parameters in the URL in React applications using the react-router-dom library.

With it you can get, modify, and add search parameters in the URL without reloading the page.

How it works:

Suppose we have a URL like: /user/:id, and it also contains a query string, for example: /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;
```

Calling setSearchParams triggers a re-render of the component where the useSearchParams hook is used. This is similar to how calling setState updates the state and triggers a re-render.

searchParams object

— Besides get(), you can also use other methods such as:

How it works:

has(key): Checks if a parameter with the specified key exists.

```
const hasName = searchParams.has('name'); // true или false
```

getAll(key): Returns an array of all values for the parameter with the given key (if the same parameter can appear multiple times in the query string).

```
const allTags = searchParams.getAll('tag'); // Например: ['react', 'router']
```

keys(): Returns an iterator over all parameter keys.

```
for (const key of searchParams.keys()) {
  console.log(key);
}
```

values(): Returns an iterator over all parameter values.

```
for (const value of searchParams.values()) {
  console.log(value);
}
```

entries(): Returns an iterator over all [key, value] pairs.

```
for (const [key, value] of searchParams.entries()) {
  console.log(key, value);
}
```

forEach(callback): Calls a callback function for each key-value pair.

```
searchParams.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});
```

Router **useLocation()**

— Hook. Gets the current URL

Router useLocation()

— It is a hook provided by React Router that allows you to get the object of the current URL.

This object contains information about the current path, query string, hash, and state passed during navigation.

When to use useLocation?

useLocation is useful in the following cases:

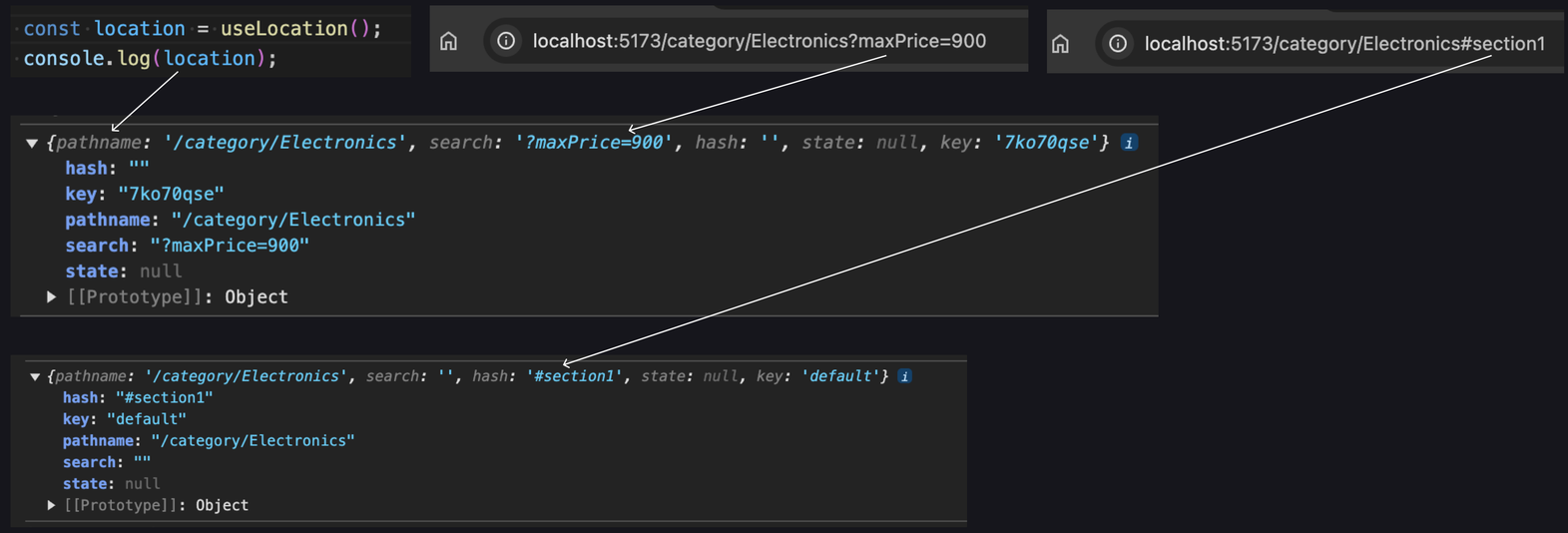
1. When you need to know the current path (pathname).
2. For working with query string parameters (search query).
3. For getting state passed via navigate (for example, additional route info).
4. For tracking path changes in the application.
5. For analytics.

```
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;
```

Router useLocation()

— It returns an object.

useLocation returns an object:



Description of all properties of the useLocation object:

Property	Description
pathname	The path of the current URL (e.g., /products/123).
search	The query string from the URL (e.g., ?category=electronics&sort=asc). (useSearchParams is the preferred way to work with query strings)
hash	The hash (URL fragment after #, e.g., #section1).
state	A state object passed during navigation (for example, via navigate or Link).
key	A unique identifier of the route (used by React Router for navigation).

Router useLocation()

— more about the state property

When to use the state property from useLocation?

Using state in React Router routes is needed for passing temporary data between pages that:

1. Should not appear in the URL.
2. Do not need global state (e.g., Redux or Context).
3. Should be used only for the current session (data is not preserved after a page reload).

```
<Link to={"/category/Electronics"} state={{ from: "Home", maxPrice: 600 }}>
  About
</Link>
```

```
  const location = useLocation();
  const maxPrice = location.state.maxPrice;
```

We took the maximum price value from the state property of the useLocation hook to filter products by maximum price.

Router **useNavigate()**

— Hook. Programmatically redirects to the specified page (in response to an event or action)

REACT JS

useNavigate()

— This is a hook from the react-router-dom library that allows you to programmatically navigate between pages in your React application.

It is useful when you need to move the user to another page in response to some action (for example, after submitting a form or clicking a button), without using the <Link> component.

How it works:

The hook returns a navigate function, which you can call to move to another page.

```
import React from 'react';
import { useNavigate } from 'react-router-dom';

function HomePage() {
  const navigate = useNavigate()  // Get the navigate function

  const goToPricing = () => {
   // Redirect the user to the Pricing page
    navigate('/pricing');
  };

  return (
    <div>
      <h1>Welcome to the Home Page!</h1>
      <button onClick={goToPricing}>Go to Pricing</button>
    </div>
  );
}
```

Back navigation: You can navigate back (similar to the browser's "Back" button):

```
navigate(-1);    // Goes back to the previous page
```

Forward navigation (in history): You can also move forward in the history if the user has already visited other pages:

```
navigate(1);    // Forward by one page in history
```

Replace current path: By default, navigate() adds a new entry to the browser history, allowing the user to go back.
If you need to replace the current path (so that the "Back" button does not return to the previous route), you can pass an extra parameter { replace: true }:

```
navigate('/dashboard', { replace: true });
```

Passing state: When navigating, you can pass additional state via an object:

```
navigate('/profile', { state: { userId: 123 } });
```

This state can then be retrieved in the target component using the useLocation hook.

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

function Profile() {
  const location = useLocation();
  const { userId } = location.state || {};
  return <div>User ID: {userId}</div>;
}
```

Router **<Navigate />**

— Component. Programmatically redirects to the specified page (automatic redirect)

Router <Navigate />

— This is a React Router component that:

• Redirects the user to the specified path.

• Can be used for redirects (for example, after authentication or on a 404 page).

How to use <Navigate />?

1. Simple redirect:

```
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": Specifies the path to redirect the user.
replace={true}: Specifies that the current route will be replaced in the browser history (a new entry will not be added).

2. Redirect via routes:

You can use <Navigate /> in the route configuration.

```
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} />,  // Redirect to the home page
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

Here path="*" means that for any unknown path the user will be redirected to /.

Router <Navigate />

— Continuation

How to use <Navigate />?

3. Redirect after performing an action:

Sometimes you need to redirect the user after performing an action, for example, after submitting a form.
After successful login, the user is redirected to the main page (/).

```jsx
import { useState } from 'react';
import { Navigate } from 'react-router-dom';

function LoginPage() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  const handleLogin = () => {
    // Authorization logic
    setIsLoggedIn(true);
  };

  if (isLoggedIn) {
    return <Navigate to="/" />;
  }

  return (
    <div>
      <h1>Login</h1>
      <button onClick={handleLogin}>Log in</button>
    </div>
  );
}

export default LoginPage;
```

4. Passing state via <Navigate />:

You can pass data during navigation using the state property:

```jsx
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;
```

Getting the state on the target page:

```jsx
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()

— What and when to use?

<Link />

<Link> — this is a React Router component that creates a navigation link, allowing the user to go to another route when clicked.

Used for declarative navigation: you define the link in JSX, and it will always lead to the specified route.

When to use <Link>?

• When the user should see and click a link to navigate.
• Example: Navigation menu, buttons to switch pages, links to other sections.

```jsx
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>
  );
}
```

Pros:

• Simplicity: adds a link to the DOM.
• Convenient for visible navigation elements.

Cons:

• Not suitable for programmatic navigation (e.g., after an action).

<Navigate />

<Navigate> — this is a React Router component that allows you to perform an automatic redirect.
When <Navigate> is rendered, it immediately redirects the user to the specified route.

When to use <Navigate>?

When a redirect should happen conditionally depending on your app logic.
Example: The user is not authorized and should be redirected to the login page.

```jsx
import { Navigate } from "react-router-dom";

function PrivateRoute({ isAuthenticated }) {
  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }
  return <div>Private Content</div>;
}
```

Pros:

• Useful for conditional redirects.
• Can pass state (state) when redirecting.

Cons:

• Can only be used in JSX, not suitable for redirects from functions or events.

useNavigate()

• useNavigate — this is a React Router hook that allows you to perform programmatic navigation (go to another route) in response to a user event or action.
• It replaces the old useHistory from React Router v5.

When to use useNavigate?

• When navigation should happen in response to a user action or after some code execution.
• Example: After successful login, submitting a form, or clicking a button.

```jsx
import { useNavigate } from "react-router-dom";

function LoginPage() {
  const navigate = useNavigate();

  const handleLogin = () => {
    // Authorization logic
    navigate("/dashboard"); // Programmatic redirect
  };

  return <button onClick={handleLogin}>Log In</button>;
}
```

Pros:

• Ideal for programmatic navigation.
• Can be used in any function or event.

Cons:

• Not suitable for declarative navigation, since it’s not part of JSX.

CSS **Modules**

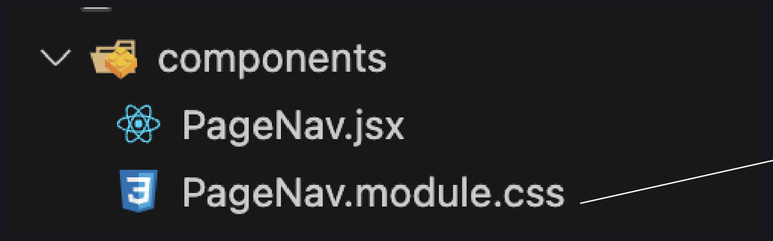
— How do they work?

CSS Modules

— This is a technique and a set of tools for working with CSS in JavaScript projects (for example, in React), which automatically isolates styles to avoid class name conflicts and styles leaking into other components.

Example of using CSS Modules

We create a separate style file for each component and define styles in it.

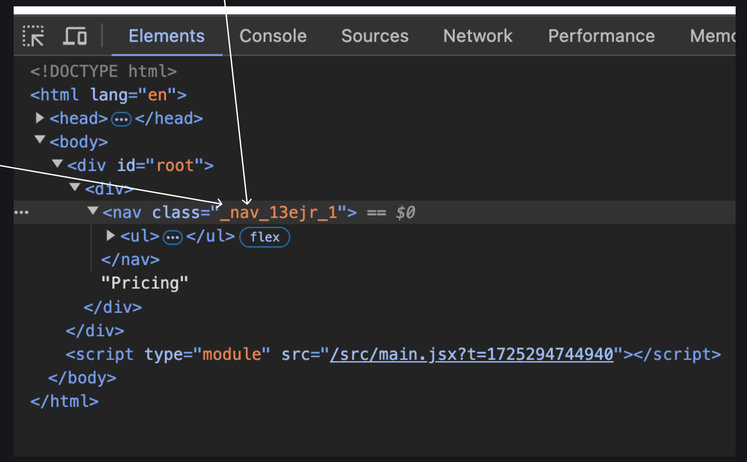



```css
.nav {
  background-color: red;
}

.nav ul {
  display: flex;
  justify-content: space-between;
}
```

Import styles into the component and use them in an element:

```jsx
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;
```

The class is applied in a modified form. This ensures the uniqueness of classes.



CSS Modules global

— CSS Modules support creating local styles with nesting, as well as defining global styles that can be used throughout the application.

Example:

```
/* styles.module.css */
:global(.global-class) {
  background-color: red;
}
.local-class {
  background-color: green;
}
```

React Rout **before** 6.4

Data Loading

— Data loading in React Router (Old approach)

REACT JS

Route Data Loading before version 6.4

Before version 6.4 in React Router, working with data in routes did not include built-in mechanisms for loading data directly into routes. Instead, developers used hooks such as useEffect, managing data state manually inside components.

— You can copy the code from the comments below and test it

Data processing flow

Regular fetch function to 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();
};
```

Site structure

```
// 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
  },
]);
```

Posts page

```
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>
  );
}
```

1. Initial rendering and page display:

• When the user navigates to the page, React renders the component. At this point, basic elements of the component (e.g., headers, static text) are displayed before data is loaded.

2. Running useEffect:

• After the first render, useEffect is triggered, since by default it executes after the component is mounted.

3. Data loading:

• Inside useEffect, asynchronous data loading occurs (e.g., using fetch), and once finished, the data is saved into state using the function returned by useState.

4. Re-rendering:

• When the state is updated (in this case, the data fetched from the API), React calls a re-render of the component to update the UI with the new data.

React Router v6.4+

loading

— Route parameter for data loading from the server (Modern approach)

Rout 6.4 + Data Loading

— This is a new version of the routing library for React applications.

It greatly simplifies route management and includes many new features compared to previous versions, such as declarative routing, support for asynchronous data via loader and action, as well as improved work with nested routes.

React Router v6.4 was released on September 13, 2022.

Main changes in React Router v6.4:

1. Support for data loading with loader.
2. Handling form data with action.
3. Centralized error handling with errorElement.
4. Improved handling of asynchronous operations.
5. New hooks for tracking navigation state (useNavigation) and error handling (useRouteError).

Rout 6.4 Data Loading (loading)

— A new way of loading data

— You can copy the code from the comments below and test it

loader — Data loading before rendering

loader is a function that is called before the component is rendered and is used to load data (for example, an API request or working with local data).
The results of this function are passed to the component via the useLoaderData hook.

```
import { createBrowserRouter, RouterProvider, useLoaderData } from "react-router-dom";

// Function for loading data
async function fetchProduct({ params }) {
  const response = await fetch(`/api/products/${params.productId}`);
  return response.json();
}

// Component for displaying data
function Product() {
  const product = useLoaderData();   // Get the data loaded through the loader
  return <h1>{product.name}</h1>;
}

  // Route setup
const router = createBrowserRouter([
  {
    path: "/product/:productId",
    element: <Product />,
    loader: fetchProduct,    // Specify the loader function
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

- loader performs an asynchronous operation (for example, loading data from the server) before rendering the component.
- The data obtained in loader is passed to the component via the useLoaderData hook.
- If an error occurs during data loading, it can be handled via errorElement (explained later).

React Router v6.4+
errorElement

— Route parameter for handling errors on a page

Rout 6.4 Data Loading (errorElement)

— It is a component that renders when an error occurs during data loading, form submission, or component rendering.

errorElement — Route Error Handling

errorElement is a component that is rendered if an error occurs during data loading, form submission, or component rendering. It helps create custom error pages.

```jsx
import { createBrowserRouter, RouterProvider, useRouteError } from "react-router-dom";

// Component for displaying an error
function ErrorPage() {
  const error = useRouteError();  // Get error information
  return (
    <div>
      <h1>Oops! Something went wrong.</h1>
      <p>{error.statusText || error.message}</p>
    </div>
  );
}

// Loader function that may throw an error
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();
}

// Route setup with error handling
const router = createBrowserRouter([
  {
    path: "/product/:productId",
    element: <Product />,
    loader: fetchProduct,
    errorElement: <ErrorPage />,  // Specify the component for error handling
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

- If an error occurs in a loader or action, the errorElement component automatically renders instead of the main component.
- Errors can be accessed using the useRouteError() hook.

REACT JS

errorElement 6.4 +

— Step-by-step workflow:

— You can copy the code from the comments below and test it.

How errorElement works?

1. When an error occurs in a route, React Router automatically replaces the standard content with the component specified in errorElement.

2. The error is passed to the component via the error prop.

Fetch function with a data request

```
// 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();
};
```

Page that uses the data

```
// 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>
  );
}
```

Page that is displayed in case of an error

```
// Component for handling errors
function ErrorBoundary({ error }) {
  return (
    <div>
      <h1>Error</h1>
      <p>{error.message || "Something went wrong"}</p>
    </div>
  );
}
```

Application structure

```
// 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
  },
]);
```

REACT JS

useRouteError()

— The useRouteError() hook is used in the route's errorElement component to get information about an error that occurred during data loading (loader) or actions (action), as well as during any code execution in the process of rendering the current route.

— You can copy the code from the comments below and test it.

How useRouteError() works:

1-Create a server request and throw an error if the server response is invalid.

```
// 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();
};
```

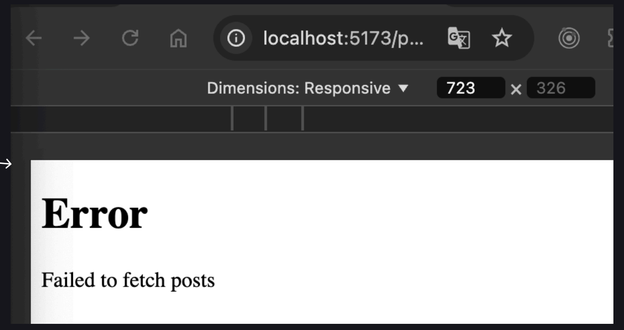
2-Define the component that will be rendered in case of an error.

```
// 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
  },
]);
```

The useRouteError() hook allows you to g et the error object that occurs during a fet ch request.

3-Declare a component to display on the screen when an error occurs.

```
// 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)**

— The useNavigation hook allows you to track the state of the current navigation.

Rout 6.4 Хук (useNavigation)

— a hook useNavigation that allows tracking the state of the current navigation

Tracking navigation state with useNavigation

useNavigation returns an object with the state of the current navigation (idle, submitting, loading).
This is useful for showing loading messages or blocking the UI during asynchronous operations.

- **idle** – no active navigation or loading.
- **loading** – data for a new route is being loaded.
- **submitting** – a form submission or other action is in progress.

Using useLoaderData and useParams

Create a component that displays the navigation state:

```
function Layout() {
  const navigation = useNavigation();

  return (
    <div>
      {navigation.state === "loading" && <p>Loading...</p>}
      <Outlet />
    </div>
  );
}
```

Wrap your entire application in this component:

```
// 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

— Route parameter for sending data to the server

REACT JS

Rout 6.4 Параметр (Action)

— Steps for usage

— You can copy the code from the comments below and test it.

action — Form Data Handling

action is used to handle data changes (for example, submitting form data via POST, PUT, DELETE, etc.).
When a form is submitted, the action function is called to perform an operation on the server (e.g., creating or updating a record).

```
import { createBrowserRouter, RouterProvider, Form } from "react-router-dom";

  // Function for handling form submission
async function createOrder({ request }) {
  const formData = await request.formData();
  const orderDetails = Object.fromEntries(formData);   // Convert form data into an object

  // Send data to the server
  await fetch("/api/orders", {
    method: "POST",
    body: JSON.stringify(orderDetails),
    headers: { "Content-Type": "application/json" },
  });
}

  // Component with a form
function CreateOrder() {
  return (
    <Form method="POST">  {/*  the form will automatically send data through action  */}
      <input type="text" name="customerName" required placeholder="Customer Name" />
      <button type="submit">Create Order</button>
    </Form>
  );
}

  // Route setup
const router = createBrowserRouter([
  {
    path: "/create-order",
    element: <CreateOrder />,
    action: createOrder,   // Specify the action function
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

1. Form in a component submits data:
• When the user fills out and submits the form, it sends data to the server (or is handled locally).
• In React Router, forms can be made declarative using the <Form> component.

2. action handles the request:
• When the form is submitted, React Router calls the action function associated with the route that the form is tied to.
• The action function receives an object containing request and params.

3. Data processing:
• inside action you can:
• Get data from the request.
• Extract route parameters (params).
• Perform an API call, update the database, or modify local state.
• Return a result that will be accessible via useActionData.

4. action result:
• Data returned from action is available in the component using the useActionData hook.

• action is called when the form is submitted. The form should use the POST, PUT, or DELETE method.
• The action function receives the request object containing form data and can send it to the server.
• Any asynchronous operation can be performed in action (e.g., API request, saving data).

Context API

— Passing props solution

Context API

— Designed to solve the “props drilling” problem, when data is passed through multiple levels of the component tree from parent to child.

Context API allows sharing data directly between components at any level of the tree, avoiding the need to pass it through every intermediate level.

Main components of Context API:

1. Creating a context (createContext)
2. Provider
3. Context consumers (useContext and Consumer)

1. Creating a context

A context is created using the createContext() function. This creates a context object that contains two components:
• Provider — provides data.
• Consumer — receives data (most often useContext is used instead of Consumer).
The function is created outside the component.

The function returns a component, so the function name is capitalized.

```
const MyContext = React.createContext();
```

2. Context Provider

The Provider is a component that wraps the component tree and provides data that can be used by any component inside it. The Provider takes a special value prop that contains the data to be passed. To use it — wrap your JSX in the return of the parent component with MyContext.Provider.

```
<MyContext.Provider value={ /* data */ }>
  { /* child components */ }
</MyContext.Provider>
```

3. Context Consumers

The useContext hook allows you to access the context data in any component inside the Provider.

```
import { useContext } from 'react';
const value = useContext(MyContext);
```

use**Reducer()**

— hook

useReducer()

— This is a React hook that allows you to manage complex component state.
It is similar to useState, but provides more capabilities for organizing and managing state, especially if state changes depend on different types of actions.

```jsx
import React, { useReducer } from 'react';

// Defining the reducer function
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() {
  // Using useReducer with an initial value of 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>
  );
}
```

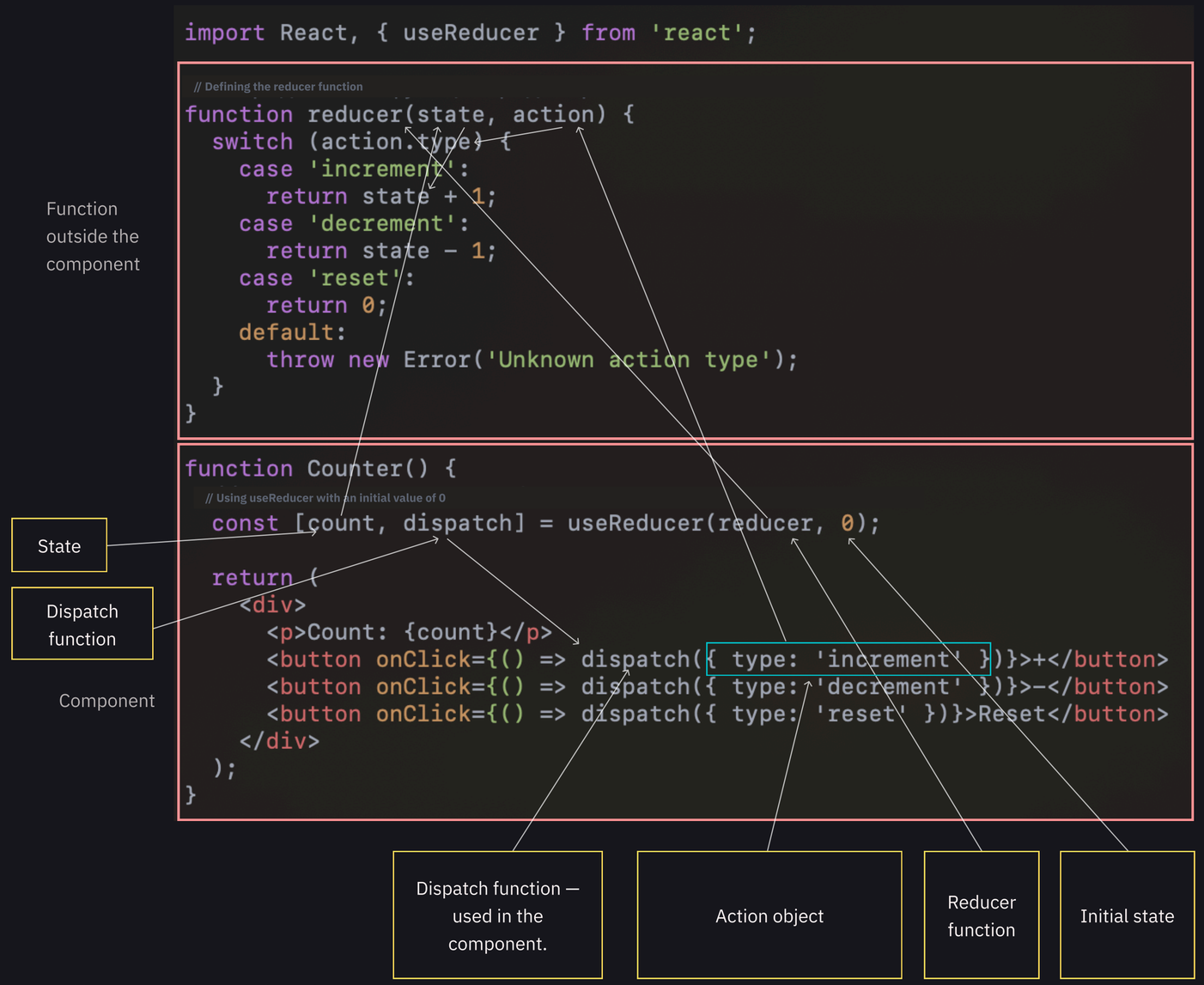
useReducer takes two arguments:

1. reducer: a reducer function defined above (outside the component).
2. { count: 0 }: the initial state.

It returns an array with two elements:

1. state: the current state (in this case — the number 0).
2. dispatch: a function to send actions to the reducer function as an argument.

useReducer() **workflow**



— You can copy the code from the comments below and test it

Step-by-step process:

1.Rendering the initial state:

• On the first render, useReducer initializes the state { count: 0 }, and the component displays Count: 0.

2. Handling button clicks:

• When the user clicks the + button, the function dispatch({ obj }) is executed.

The dispatch function sends an “action” object to the external reducer function, which takes the type property of the action object and checks it in a switch statement.

If action.type == "increment" → increases the state by 1.
If action.type == "decrement" → decreases the state by 1.
If action.type == "reset" → resets count to 0.

3. Updating the component:

• After each dispatch call, React re-renders the component with the new state, and the updated count value is displayed on the screen.

useReducer() **payload**

— action.payload

useReducer() workflow

Function outside the component

```jsx
// Defining the reducer function
function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return state + 1;
    case "decrement":
      return state - 1;
    case "reset":
      return 0;
    case "increment_by":
      // Use the value from action.payload
      return state + action.payload;
    default:
      throw new Error("Unknown action type");
  }
}
```

```jsx
function Counter() {
  // Using useReducer with an initial value of 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>

      {/* Button with payload, increment the counter by 5 at once */}
      <button onClick={() => dispatch({ type: "increment_by", payload: 5 })}>+5</button>
    </div>
  );
}

export default Counter;
```

State

Dispatch function

Reducer function

Component

Using the dispatch function:

Action object and dispatch function

Initial state

Additional action data

— You can copy the code from the comments below and test it.

Step-by-step process:

1.Initial state rendering:

• On the first render, useReducer initializes the state { count: 0 }, and the component displays Count: 0.

2. Handling button clicks:

• When the user clicks the + button, the function dispatch({ obj }) is executed.

The dispatch function sends the "action" object to the external reducer function, which reads the type property of the action object and checks it in a switch.

If action.type == "increment", it increases the state by 1.
If action.type == "decrement", it decreases the state by 1.
If action.type == "reset", it resets count to 0.

If the +5 button is clicked, the state increases by the number specified in action.payload.

3. Component update:

• After each dispatch call, React re-renders the component with the new state, and the updated count value is displayed on the screen.

payload — is "extra information" about what exactly should be done with the state.

useReducer() **initialState**

+ e.target.value

Initial state as an object, not a primitive

```
// 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
};
```

Reducer function for state management

```
// 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");
  }
}
```

Component

```
function Counter() {
  // 3. 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. onChange event:

When the user types a value into the input field, the onChange event is triggered and calls the dispatch function.

- Action type: "update_input"
- The value entered in the field is passed via action.payload.

2. Binding the value state:

The input field is bound to state.inputValue via value.
This means that every time state.inputValue updates, the input field’s value is automatically synchronized.

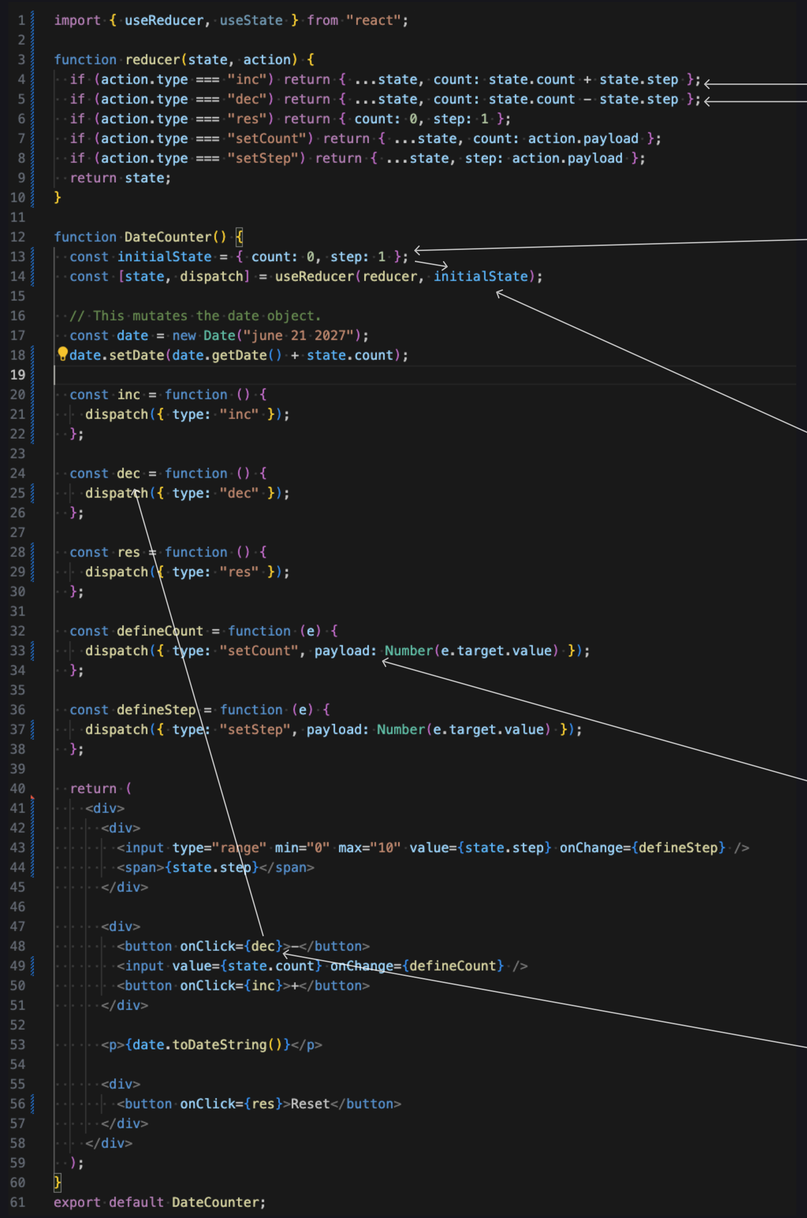
3. Clearing the field:

After clicking the "Increase by" button, inputValue is cleared so the user sees an empty field.
This is done via dispatch with type "update_input" and an empty value (payload: "").

initialState :

The initialState object defines the initial values for all state variables.
It is used as the base state on the first render of the component.

A more realistic **example of using useReducer()**



```jsx
import { useReducer, useState } from "react";

function reducer(state, action) {
  if (action.type === "inc") return { ...state, count: state.count + state.step };
  if (action.type === "dec") return { ...state, count: state.count - state.step };
  if (action.type === "res") return { count: 0, step: 1 };
  if (action.type === "setCount") return { ...state, count: action.payload };
  if (action.type === "setStep") return { ...state, step: action.payload };
  return state;
}

function DateCounter() {
  const initialState = { count: 0, step: 1 };
  const [state, dispatch] = useReducer(reducer, initialState);

  // This mutates the date object.
  const date = new Date("june 21 2027");
  date.setDate(date.getDate() + state.count);

  const inc = function () {
    dispatch({ type: "inc" });
  };

  const dec = function () {
    dispatch({ type: "dec" });
  };

  const res = function () {
    dispatch({ type: "res" });
  };

  const defineCount = function (e) {
    dispatch({ type: "setCount", payload: Number(e.target.value) });
  };

  const defineStep = function (e) {
    dispatch({ type: "setStep", payload: Number(e.target.value) });
  };

  return (
    <div>
      <div>
        <input type="range" min="0" max="10" value={state.step} onChange={defineStep} />
        <span>{state.step}</span>
      </div>

      <div>
        <button onClick={dec}>-</button>
        <input value={state.count} onChange={defineCount} />
        <button onClick={inc}>+</button>
      </div>

      <p>{date.toDateString()}</p>

      <div>
        <button onClick={res}>Reset</button>
      </div>
    </div>
  );
}
export default DateCounter;
```

Destructuring an object and updating one of its properties

A variable with the initial state

The initial state is an object, not a primitive

payload is the commonly accepted name for the property used when updating an object in the reducer function

Use a separate function where dispatch is called

Redux

— State management in a large project

Redux

— In structure, it is very similar to how the useReducer hook works.

Main concepts (terms/blocks) of Redux:

1. **Store** — a single state object for the entire application. It contains all the application state.
2. **Actions** — objects that describe events or changes that should happen in the state.
3. **Reducers** — pure functions that take the current state and an action, then return a new state.
4. **Dispatch** — a way to send an action to the store so the reducer can process it and update the state.
5. **Selectors** — functions that extract the necessary data from the state for use in components.

Main steps for creating and using Redux

1. Installing Redux and React-Redux:
First, install the redux and react-redux libraries.

```
npm install redux react-redux
```

2. Creating a Reducer:
A reducer is a pure function that takes the current state and an action, and returns a new state based on the action type. (Just like in 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

— steps for creation and usage

3. Creating the Store

The Store is the central state storage of the application. It is created using the createStore function.

```
// store.js

import { createStore } from 'redux';
import rootReducer from './rootReducer';

// Create a store with reducers
const store = createStore(rootReducer);

export default store;
```

4. Connecting Redux to React

To connect React to Redux, the Provider component from the react-redux library is used. It provides access to the store for all components in the application.

```
// 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. Creating Actions

Actions are objects that are sent to the store to trigger a state change. They always have a type field and sometimes additional information in payload.
(Just like in useReducer).

```
// actions.js

// Action for depositing funds
export const deposit = (amount) => {
  return {
    type: 'account/deposit',
    payload: amount,
  };
};

// Action for withdrawing funds
export const withdraw = (amount) => {
  return {
    type: 'account/withdraw',
    payload: amount,
  };
};
```

Redux

— steps for creation and usage

6. Accessing state in components with useSelector

To access data from the Redux store in a component, the useSelector hook is used.

```
// 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. Dispatching actions with useDispatch

To change the state, the useDispatch hook is used, which allows sending an action to the store.

```
// 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

— steps for creation and usage

8. Combining reducers (combineReducers)

If the application has several independent parts of the state, you can use the combineReducers function to merge multiple reducers.

```
// rootReducer.js

import { combineReducers } from 'redux';
import { accountReducer } from './reducers';

const rootReducer = combineReducers({
  account: accountReducer,
    // you can add more reducers if needed
});

export default rootReducer;
```

9. Viewing the current state

To view the current state of the application (for example, for debugging purposes), you can use the getState method from the store.

```
console.log(store.getState());
```

Full example:

```
// 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**

— Asynchronicity

Redux Thunk

— This is middleware for Redux that allows writing asynchronous actions. It provides the ability to create actions in the form of functions that can perform asynchronous requests and then dispatch regular actions with the results of these requests.

Main concepts (terms/blocks) of Redux

Asynchronous actions: redux-thunk allows creating actions as functions instead of plain objects.
These functions can perform any asynchronous operations (for example, server requests).

Access to dispatch and getState: Inside the action function provided by Thunk, two arguments are available: dispatch to send actions and getState to get the current store state.

Main steps for using Redux Thunk:

1. Installing Redux Thunk
First, you need to install the library:

```
npm install redux-thunk
```

2. Connecting Thunk to the store
Thunk must be connected as middleware when creating the Redux store using the applyMiddleware function.

```
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

— steps for creation and usage

3. Creating asynchronous actions

Now you can create asynchronous actions using Thunk. Instead of returning an object with a type field, you can return a function that will perform asynchronous requests and then dispatch actions.

```
// actions.js

// Asynchronous action to fetch data
export const fetchUserData = () => {
  return async (dispatch) => {
    dispatch({ type: 'user/fetchStart' }); // Dispatch loading-start action

    try {
      const response = await fetch('https://api.example.com/user');
      const data = await response.json();

      dispatch({ type: 'user/fetchSuccess', payload: data }); // Successful result
    } catch (error) {
      dispatch({ type: 'user/fetchError', payload: error.message }); // Error handling
    }
  };
};
```

4. Handling states in the Reducer

To handle loading, success, or error states in the reducer, you need to add new action types.

```
// 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

— steps for creation and usage

5. Dispatching an asynchronous action from a component

Now you can use the asynchronous action in a component through the useDispatch hook.

```
// 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()); // Fetch data when the component mounts
  }, [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. Using getState

Inside an asynchronous action, you can use getState to get the current state of the application and use it for decision-making.

```
// actions.js

export const fetchIfNotLoaded = () => {
  return (dispatch, getState) => {
    const state = getState();
    if (!state.user.user) {
      dispatch(fetchUserData());   // Fetch data if it hasn't been loaded yet
    }
  };
};
```

Full example:

```
// 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**

— Modern and simplified version of Redux

Redux Toolkit

— This is a set of tools for working with Redux that simplifies many aspects of development.

It helps reduce boilerplate code, provides built-in support for asynchronous actions, and improves performance.

RTK is the officially recommended library for working with Redux.

Main functions of Redux Toolkit:

1. **configureStore** — for creating the store with built-in enhancements.
2. **createSlice** — for simplified creation of reducers and actions.
3. **createAsyncThunk** — for handling asynchronous requests.
4. **createEntityAdapter** — for working with normalized data (e.g., lists of objects).
5. **Immer** — allows working with immutable state in a convenient way.
6. **Integration with DevTools** and redux-thunk support by default.

1. Installing Redux Toolkit

First, you need to install Redux Toolkit and React-Redux (if you are working with React):

```
npm install @reduxjs/toolkit react-redux
```

npm i @reduxjs/toolkit react-redux

Redux Toolkit

— steps for creation and usage

2. Creating a store with configureStore

In classic Redux you would write something like this:

```
import { createStore, applyMiddleware } from 'redux'
import rootReducer from './reducers'
import thunk from 'redux-thunk'

const store = createStore(rootReducer, applyMiddleware(thunk))
export default store
```

And also configure DevTools separately using composeWithDevTools().

In RTK

The configureStore function in Redux Toolkit:

1. Automatically connects Redux Thunk as middleware.
2. Automatically enables Redux DevTools in development mode.
3. Simplifies the setup of additional middleware.
4. Under the hood, calls combineReducers if you pass an object with reducers.

```
// 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,
  },
  // Optional settings
  devTools: process.env.NODE_ENV !== 'production',
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: false,  // for example, if you need to disable something
    }),
})
```

Redux Toolkit

— steps for creation and usage

3. createSlice: simplified creation of reducer and actions

In classic Redux you would write something like:

```
// 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
```

In RTK: createSlice solves the task in one go:

1. Generates action constants and action creators.
2. Creates a reducer where we write "mutable" code (Immer wraps it and ensures immutability).
3. No need for Switch and default case constructions.

```
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  todos: [],
}

const todosSlice = createSlice({
  name: 'todos',
  initialState,
  reducers: {
    addTodo(state, action) {
      // RTK and Immer allow you to "write" to the state,
      // while actually creating immutable structures under the hood
      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 is a full-fledged reducer that we connect to configureStore.

Redux **DevTools Extention**

— A browser plugin for debugging Redux applications

Redux DevTools Extention

— это плагин для браузера, который:

1. Allows you to view the current state of the store.

2. Tracks actions and their order.

3. Enables "time travel debugging" to see how the application state has changed over time.

4. Simplifies error searching and analysis of how actions affect the state.

Separate functionality from Redux or RTK

Redux DevTools Extension exists as a separate browser extension and is not directly included in Redux or Redux Toolkit code.

Classic Redux c Redux DevTools Extention

If for some reason you are not using RTK, you can still connect Redux DevTools separately by using the redux-devtools-extension package and an enhancer when creating the store with createStore:

```
import { createStore } from 'redux'
import rootReducer from './reducers'
import { composeWithDevTools } from 'redux-devtools-extension'

const store = createStore(
  rootReducer,
  composeWithDevTools()
)
```

And install the npm package: npm install @redux-devtools/extension.

RTK c Redux DevTools Extention

In Redux Toolkit there is built-in integration with Redux DevTools via the configureStore function. If you create a store through configureStore, it can "communicate" with Redux DevTools by default (if the extension is installed and opened in the browser).

```
import { configureStore } from '@reduxjs/toolkit'
import rootReducer from './reducers'

const store = configureStore({
  reducer: rootReducer,
  devTools: process.env.NODE_ENV !== 'production',
})
```

Redux **Toolkit createAsyncThunk**

— Modern execution of asynchronous actions

Redux Toolkit

— steps for creation and usage

4. createAsyncThunk: simplified work with asynchronous requests

In plain Redux, if you need to make a request to the server (or any asynchronous operation), you would:

1. Write three actions — REQUEST, SUCCESS, ERROR.
2. Write a "thunk" (function) that calls fetch / axios, and inside it manually dispatch these three actions.

That’s a lot of boilerplate code.
createAsyncThunk automates all of this and creates three actions (pending/fulfilled/rejected) for you.

a) Declare an asynchronous "task"

```
import { createAsyncThunk } from '@reduxjs/toolkit'

// 'users/fetchUsers' — this is the prefix for actions
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async () => {
    const response = await fetch('https://jsonplaceholder.typicode.com/users')
    // return the value that will become action.payload when fulfilled
    return await response.json()
  }
)
```

When calling dispatch(fetchUsers()), Redux Toolkit automatically creates three actions:

1. users/fetchUsers/pending (request started)
2. users/fetchUsers/fulfilled (request completed successfully, data in payload)
3. users/fetchUsers/rejected (request "failed" with an error)

You don’t write these actions manually. createAsyncThunk does it for you.

Redux Toolkit

— steps for creation and usage

b) Handling the result in a slice

To react to these actions (for example, change state.loading, state.error, put data into state.users), Redux Toolkit uses extraReducers:

```
import { createSlice } from '@reduxjs/toolkit'
import { fetchUsers } from './thunks'  // The code where we declared fetchUsers

const userSlice = createSlice({
  name: 'users',
  initialState: {
    users: [],
    loading: false,
    error: null,
  },
  reducers: {}, // Regular (synchronous) reducers, if needed
  extraReducers: (builder) => {
      // .pending -> when the request started
    builder.addCase(fetchUsers.pending, (state) => {
      state.loading = true
      state.error = null
    })
      // .fulfilled -> when the request completed successfully
    builder.addCase(fetchUsers.fulfilled, (state, action) => {
      state.loading = false
      state.users = action.payload    // Here lies the data from the server
    })
      // .rejected -> when an error occurred
    builder.addCase(fetchUsers.rejected, (state, action) => {
      state.loading = false
      state.error = action.error.message    // or action.payload, if we use rejectWithValue
    })
  },
})

export default userSlice.reducer
```

- When you dispatch(fetchUsers()), Redux Toolkit automatically sends users/fetchUsers/pending.
- In extraReducers, .pending is caught and sets state.loading = true.
- When the request finishes, Toolkit sends users/fetchUsers/fulfilled (if success) or users/fetchUsers/rejected (if error).
- In .fulfilled you put action.payload (data from the server) into state.users.
- In .rejected you handle action.error or action.payload.

You don’t manually write REQUEST, SUCCESS, FAILURE actions — all this logic is built into createAsyncThunk and extraReducers.

Redux Toolkit

— steps for creation and usage

c) Using it in a React component

Usually it looks like this:

```
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())  // request to the server
  }, [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. On the first render, useEffect calls dispatch(fetchUsers()).

2. fetchUsers() is invoked and dispatches users/fetchUsers/pending. The reducer sets loading = true.

3. When the response is received, users/fetchUsers/fulfilled is dispatched, the reducer puts the data into state.users and sets loading = false.

4. The component re-renders, showing the list of users.

Error handling

createAsyncThunk

— Modern execution of asynchronous actions

Redux Toolkit

— steps for creation and usage

There are 2 ways to handle and receive an error

Option 1: throwing an error (throw new Error)

If inside your createAsyncThunk you do:

```
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async (arg, { getState, dispatch }) => {
    const response = await fetch('https://jsonplaceholder.typicode.com/users')

    // Check if the response status is OK
    if (!response.ok) {
      // If the status is not OK, just throw an error
      throw new Error(` Error! Status ${response.status}`)
    }

    const data = await response.json()
    return data
  }
)
```

Then in your extraReducers you specify:

```
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
      // If we "threw" an error, it will end up in action.error.message
      state.error = action.error.message
    })
}
```

In this case, if throw new Error(...) is triggered, Redux Toolkit will "catch" this error and dispatch the fetchUsers.rejected action with the field action.error.message = 'Error text'.

// Check if the response status is OK

REACT JS

Redux Toolkit

— steps for creation and usage

There are 2 ways to handle and receive an error

2. Option 2: using rejectWithValue (special handling)

Sometimes you may want to pass some additional data into the error, not just a message. In that case, use rejectWithValue.

```
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async (_, { rejectWithValue }) => {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/users')
      if (!response.ok) {
        // Here we don't throw an error, but return a "processed" text or object
        return rejectWithValue(`Error while loading. Status: ${response.status}`)
      }
      const data = await response.json()
      return data
    } catch (err) {
      // A network error, for example
      return rejectWithValue(err.message || 'Unknown error')
    }
  }
)
```

Then in your 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
      /*
        If we use rejectWithValue → the error text will be in action.payload. Otherwise, if we threw an error →
        check action.error.message.
      */
      if (action.payload) {
        // A handled custom error
        state.error = action.payload
      } else {
        // A thrown error (throw new Error)
        state.error = action.error.message
      }
    })
}
```

This way, by using rejectWithValue, you decide what exactly to put into the action.payload in case of an error.

And then in rejected you check:

- if action.payload exists — it means the error came from rejectWithValue;
- otherwise, look at action.error.message (a thrown error).

Additional features of **createAsyncThunk**

— Modern execution of asynchronous actions

Redux Toolkit

— steps for creation and usage

General form of the function in createAsyncThunk

In the callback you pass as the second argument to createAsyncThunk, there can be two parameters:

```
export const someThunk = createAsyncThunk(
  'something/someThunk',
  async (arg, thunkAPI) => {
    // ...
  }
)
```

The first parameter (here called arg) — this is the value you pass when calling the action:

```
dispatch(someThunk(42))
// then inside it will be arg = 42
```

The second parameter (here called thunkAPI) — this is an object that gives access to additional Redux Toolkit methods. The most useful are:

- dispatch — to dispatch other actions inside the thunk.
- getState — to look at the current Redux state.
- rejectWithValue — to return a "special" value in action.payload in case of an error.

Why do people often write (, { rejectWithValue })

“_” instead of “arg”

If you don’t use the first parameter (argument), it is often called _ or __ to make it clear: “I don’t use it.” This is not a special word, just a convention. JavaScript allows you to name the parameter anything — _, or banana.

{ rejectWithValue } instead of thunkAPI

If you only need the rejectWithValue method from thunkAPI, developers use destructuring:

```
async (_, { rejectWithValue }) => {
  // ...
}
```

Redux Toolkit

— steps for creation and usage

The second parameter (here called thunkAPI)

When you write a callback for createAsyncThunk, the second parameter is conventionally called thunkAPI. It provides access to several methods: dispatch, getState, rejectWithValue, fulfillWithValue, etc.

```
export const myAsyncThunk = createAsyncThunk(
  'someSlice/myAsyncThunk',
  async (arg, thunkAPI) => {
     // Here you can:
     // 1. Dispatch any actions:
    thunkAPI.dispatch(someAction())

     // 2. Get the current Redux state
    const state = thunkAPI.getState()
    console.log('Current state:', state)

     // Example of an asynchronous request
    const response = await fetch('https://jsonplaceholder.typicode.com/users')
    const data = await response.json()

     // Return the data
    return data
  }
)
```

Note:

1. arg — this is what you pass into dispatch(myAsyncThunk(argValue)).
2. thunkAPI — contains methods:

• dispatch: allows you to dispatch other actions inside this thunk.
• getState: returns the current Redux state.
• rejectWithValue: to pass a "special error" into action.payload when rejected.
• fulfillWithValue: a rarer case, the equivalent for a "successful" return.
• requestId, signal, abort: advanced features for canceling a request or linking to AbortController.

```
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async (arg, { dispatch, getState, rejectWithValue }) => {
     // For example: check something in the state
    const { users } = getState().usersData
    if (users.length > 0) {
      // users already exist → do not fetch again
      return users
    }

    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/users')
      if (!response.ok) {
        // We can dispatch an additional action here:
        dispatch(showNotification( Error loading users '))
        return rejectWithValue( Status: ${response.status}`)
      }
      const data = await response.json()
      return data
    } catch (error) {
      dispatch(logError(error))  // Logging, for example
      return rejectWithValue(error.message ||  'Unknown error'  )
    }
  }
)
```

Redux Toolkit

— steps for creation and usage

5. Working with normalized data with createEntityAdapter

createEntityAdapter is useful for working with collections of data (for example, lists of objects), providing easy addition, updating, and deletion of items. It automatically normalizes data and manages collections through standard methods.

```
import { createSlice, createEntityAdapter } from '@reduxjs/toolkit';

const usersAdapter = createEntityAdapter();

// Initial state with adapter
const initialState = usersAdapter.getInitialState();

const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    addUser: usersAdapter.addOne,    // Add one user
    updateUser: usersAdapter.updateOne,  // Update one user
    removeUser: usersAdapter.removeOne,  // Remove user
  },
});

export const { addUser, updateUser, removeUser } = usersSlice.actions;
export default usersSlice.reducer;
```

- *Adapters create convenient methods for working with data collections.*
- *These methods reduce the amount of code and make it more expressive.*

6. Connecting the store to React via Provider

As in plain Redux, to use the store in a React application you need to connect it via the Provider component.

```
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. Integration with DevTools

configureStore includes Redux DevTools support by default, so you don’t need additional setup to use them in development.

If you want to disable DevTools in production, you can do it like this:

```
const store = configureStore({
  reducer: rootReducer,
  devTools: process.env.NODE_ENV !== 'production',
});
```

UseMemo

— Application optimization

useMemo()

— this is a hook that memoizes the result of a function calculation. This means that if the dependencies do not change, useMemo returns the previously computed value instead of performing the computation again on every render.

— You can copy the code from the comments below and test it.

Difference from memo()

Test the code from the comments.

1 — Increase count by clicking the button
2 — Show articles by clicking the button
3 — Increase count by clicking the button

You will see that the problem remains, even though we used the memo() wrapper.

The problem remains,,
because this time we used an object as props.

And as we know, if, for example, you compare two objects with the same data, you will get false.
Objects will not be equal, since they are compared by reference, not by value.

Example: ({} != {})

Why doesn’t memo() work in this case?

The memo() function prevents a component from re-rendering if its props haven’t changed. However, in the example this doesn’t work because an archiveOptions object is passed as a prop.
Objects in JavaScript are compared by reference, not by value.
This means that every time the App component renders, a new archiveOptions object is created, even if its contents remain unchanged.

In such a case, memo() thinks the props have changed because it compares object references (not their contents), and therefore the Archive component will re-render every time the App component re-renders.

useMemo()

— To avoid creating a new object on every render, you can use useMemo.

It memoizes the result of a calculation and keeps the object as long as its dependencies haven’t changed.

— You can copy the code from the comments below and test it.

Difference from memo()

Test the code from the comments:

1 — Increase count by clicking the button
2 — Show articles by clicking the button
3 — Increase count by clicking the button

You will see that the problem is solved, even though we are still using an object as props.

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

  // Using useMemo to memoize archiveOptions
  const archiveOptions = useMemo(
    () => ({
      show: false,
      title: "Archived Posts",
    }),
    []
  );

  return (
```

How does useMemo() work in this case?

To avoid creating a new object on every render, you can use useMemo. It memoizes the result of the calculation and keeps the object as long as its dependencies haven’t changed. In this case, you can memoize archiveOptions so that the object is not recreated on every render.

REACT JS

useMemo()

— More details

— You can copy the code from the comments below and test it

useMemo()

— is a hook that memoizes the result of a function and returns it only when the dependencies change.
Its syntax looks as follows:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

• **First argument:** This is a function that returns a value.
This can be a computation that takes a lot of time or resources.

• **Second argument:** The dependency array. This array tells useMemo when to recompute the value.
If none of the dependencies have changed, useMemo will return the previously memoized (saved) value

Syntax of useMemo()

```
const memoizedValue = useMemo(() => {
    // function that returns the result
  return сложные_вычисления(a, b);
}, [a, b]);  // dependency array
```

How does the dependency array work?

The dependency array is a list of variables on which the result of the function in useMemo depends. If the values of the variables in the array change during a component render, useMemo will recalculate the memoized value; otherwise, it will return the previously computed result.

1. If a dependency changes: The function passed to useMemo will be executed again, and the new value will be returned and remembered.
2. If a dependency does not change: useMemo will simply return the previously memoized value and will not call the function again.

Empty array []: If you pass an empty dependency array, then the function passed to useMemo will be executed only once — when the component is mounted. This is similar to the behavior of useEffect with an empty dependency array:
In this case, the function will run only once on mount, and the result will be used until the component is unmounted.
TEST THE CODE FROM THE COMMENT

UseCallback

— Application optimization

REACT JS

useCallback()

— It is a React hook that returns a memoized version of a function, keeping the same reference to the function between renders if the dependencies (listed in the dependency array) have not changed.

This is useful to prevent unnecessary function recreations on every render.

— You can copy the code from the comments below and test it.

How does useCallback work?

When a component renders, new versions of functions are created each time. If these functions are passed through props, the child component will re-render, even if its props have not actually changed.
useCallback solves this problem by memoizing the function and preventing its recreation if dependencies do not change.

Syntax of useCallback:

```
const memoizedCallback = useCallback(() => {
  // function logic
}, [dependencies]);
```

- The first argument — the function that will be memoized.
- The second argument — the dependency array. If any dependency changes, the function is recreated.

TEST TWO VERSIONS OF THE CODE FROM THE COMMENT

Example 1: Without useCallback (Performance problem)

In this example, every time the component state changes, the function is recreated, which causes the child component to re-render, even if its props have not changed.

Problem:

- Every time you click the “Increase Count” button, the App component renders, and the handleClick function is recreated.
- This causes unnecessary re-renders of the PostList component, even though its props have not changed, which worsens performance with a large number of posts.

Example 2: With useCallback

- Now the handleClick function is memoized with useCallback. It is created once during the first render of the App component and is not recreated when the count state changes.
- This prevents unnecessary re-renders of the PostList component, since its props no longer change on every render.

useCallback()

— More details

useCallback()

— is a hook in React that is used for memoizing functions.
It returns the same function if its dependencies do not change. This is useful to prevent unnecessary recreation of functions on every render, especially if the function is passed as a prop to a child component, which could otherwise cause extra re-renders.

When to use useCallback?

- When a function depends on variables (for example, state or props) and you want to control when it should be recreated.
- When you have a complex or resource-intensive function that should not be recreated on every render of the component.

```
const memoizedCallback = useCallback(() => {
    // function logic
}, [dependency1, dependency2]);
```

- The first argument — the function you want to memoize.
- The second argument — the dependency array. If at least one of the dependencies changes, useCallback creates a new version of the function.

How does the dependency array work?

The dependency array in useCallback specifies which values the function depends on. The function is recreated only if one of the dependencies in this array changes. If the dependencies remain unchanged, useCallback returns the same function that was created earlier.

Boundle & LazyLoad

— Application optimization

Bundle Splitting

— this is an optimization technique in which a large JavaScript bundle is split into several smaller parts.

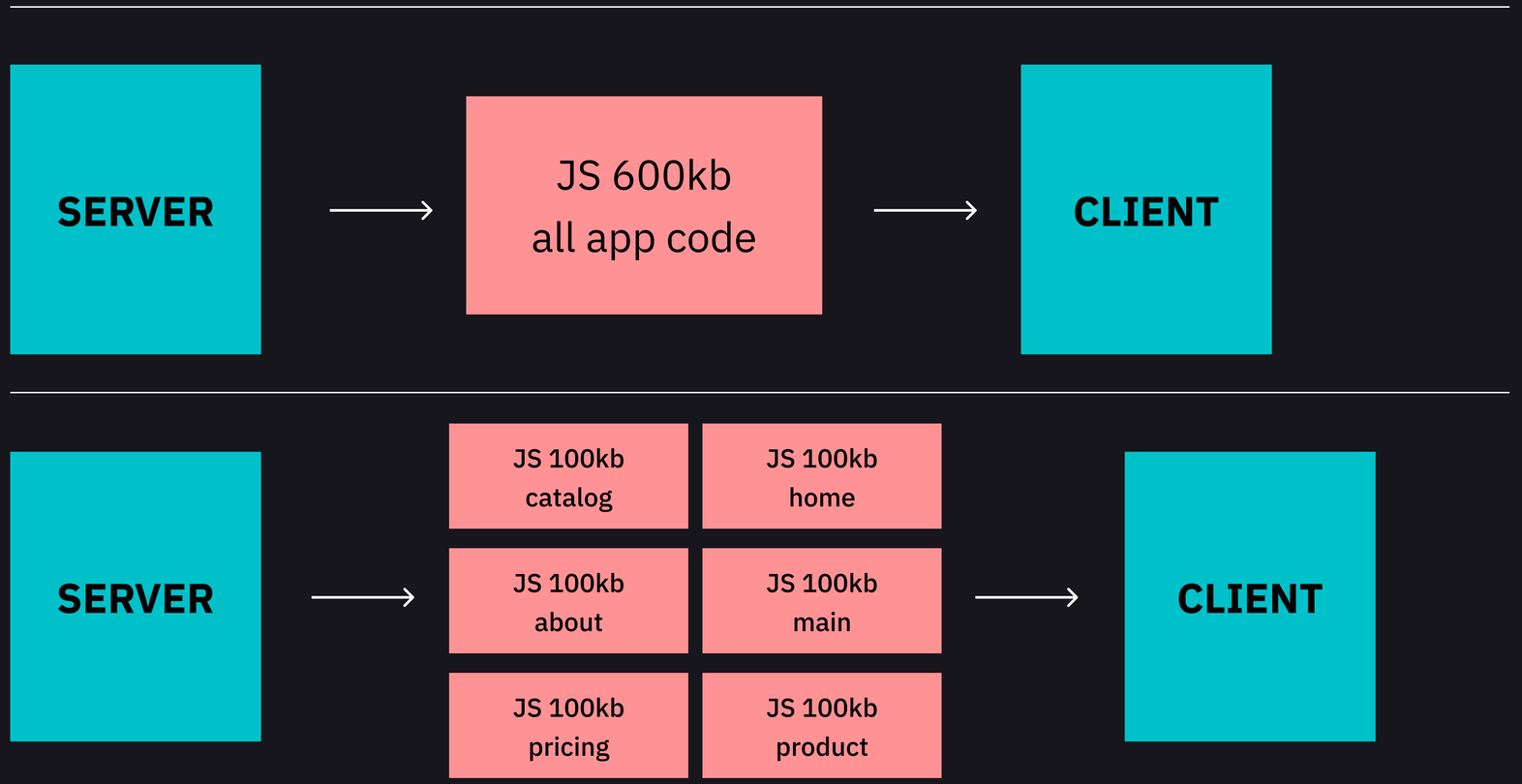
This allows loading only those parts of the application that are necessary for the current view, instead of loading all the code at once.

Without bundle splitting:

The application loads all the code at once, even if some components will not be used immediately.

With bundle splitting:

The application loads only what is needed for the current screen, and the remaining code is loaded only when necessary (for example, when the user navigates to another page or interacts with a specific part of the application).



Lazy Loading

— this is a technique that allows loading the component code only when it is actually needed (for example, when the component should be displayed on the screen).

Lazy Loading in React

React supports lazy loading of components using the built-in React.lazy() hook.

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

// Lazy import of the component
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <h1>Welcome to the app</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent /> {/*  This component will only be loaded when it is rendered  */}
      </Suspense>
    </div>
  );
}

export default App;
```

In this example, the LazyComponent is loaded only when it needs to be rendered.
Suspense is a component that allows you to specify fallback content (for example, “Loading...”) that will be shown while the lazy component is loading.

How are Bundle Splitting and Lazy Loading related?

Both of these methods work together to optimize the loading of JavaScript bundles in a React application:

- Bundle splitting allows you to divide the code into smaller parts (bundles), each of which can be loaded separately.
- Lazy loading uses these split bundles to load them only when they are needed.

Lazy loading can be applied both to pages and to components

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

// Lazily load the component
const MyComponent = React.lazy(() => import('./MyComponent'));

function App() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <MyComponent /> {/*  This component will only be loaded when needed  */}
      </Suspense>
    </div>
  );
}

export default App;
```

Image **hashing**

— Preparing images for deployment

Images

— Place images via import into the respective component.

Image hashing

It is recommended in Vite (as well as in most modern bundlers) to import an image and use it as a module so that Vite can optimize/hash it.

```
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;
```

When we say that the bundler (Vite, Webpack, etc.) “hashes” files, it means that in the final bundle (dist folder) the files (images, JS, CSS, etc.) get generated “suffixes” in their names — usually a set of letters and numbers, for example:

```
logo.svg → logo.abcd1234.svg
```

This “hash” (often MD5 or another algorithm) changes when the file itself changes and serves two main purposes:

1. **Unique filename.** If the user (or proxy server) cached the file under the old name, then after the next build, when the file content has changed, it will already have a different name (with a different hash). Thanks to this, the browser understands that the file is new and loads it again (instead of fetching it from cache).

2. **Cache-busting.** Without a hash, browsers may not notice changes, for example, if aggressive caching is enabled. Then users may not see the new versions right away. With a hash in the filename, any change in the file is guaranteed to produce a new name.

NPM **Run Build**

— Project build with Vite

NPM **Run Build**

— The command npm run build in a Vite + React project is used to create an optimized version of your application, ready for deployment to a production server.

What the command npm run build does:

Creates an optimized bundle:

• Vite collects all the resources of your application (HTML, CSS, JavaScript, images) and combines them into a minimized and production-optimized bundle.
• The bundle includes only those parts of the code that are actually used in the application (tree-shaking).

Minifies the code:

• JavaScript and CSS are minimized (spaces, comments, and unnecessary characters are removed) to reduce file size.
• The esbuild library is used for fast minification.

Optimizes loading:

• Generates links for lazy loading (dynamic imports) in your modules.
• Asynchronous modules are loaded only when they are needed.

Creates the dist folder:

• All built and optimized files are placed in the dist folder, which is ready to be uploaded to the server.

Optimizes static assets:

• Images (e.g., .png, .jpg, .svg) can be optimized if you use appropriate plugins (such as vite-imagemin).
• Files from the public/ folder are copied into the dist folder without changes.

Project deployment

— Hosting the website

Run the script and upload the files to the hosting

— Deploying project files to the server manually

“Local build + manual upload”

1. Run npm run build locally.
2. Take the contents of the dist/ folder.
3. Go to your hosting control panel (for example, ISPmanager, cPanel, or something else) and, using the built-in file manager or SFTP/FTP, upload all the files from dist/ to the root directory of the site (or to the required subdirectory).
4. Configure the domain to point to this folder (if necessary).